Available online at www.sciencedirect.com
ScienceDirect
Comput. Methods Appl. Mech. Engrg. 373 (2021) 113552
Computer methods in applied mechanics and engineering
www.elsevier.com/locate/cma

SciANN: A Keras/TensorFlow wrapper for scientific computations and physics-informed deep learning using artificial neural networks

Ehsan Haghighat*, Ruben Juanes

Massachusetts Institute of Technology, Cambridge, MA, United States of America

Received 11 May 2020; received in revised form 16 October 2020; accepted 27 October 2020
Available online 26 November 2020

Abstract

In this paper, we introduce SciANN, a Python package for scientific computing and physics-informed deep learning using artificial neural networks. SciANN uses the widely used deep-learning packages TensorFlow and Keras to build deep neural networks and optimization models, thus inheriting many of Keras's functionalities, such as batch optimization and model reuse for transfer learning. SciANN is designed to abstract neural network construction for scientific computations and solution and discovery of partial differential equations (PDE) using the physics-informed neural networks (PINN) architecture, therefore providing the flexibility to set up complex functional forms. We illustrate, in a series of examples, how the framework can be used for curve fitting on discrete data, and for solution and discovery of PDEs in strong and weak forms. We summarize the features currently available in SciANN, and also outline ongoing and future developments.

Keywords: SciANN; Deep neural networks; Scientific computations; PINN; vPINN

1. Introduction

Over the past decade, artificial neural networks, also known as deep learning, have revolutionized many computational tasks, including image classification and computer vision [1–3], search engines and recommender systems [4,5], speech recognition [6], autonomous driving [7], and healthcare [8] (for a review, see, e.g. [9]). Even more recently, this data-driven framework has made inroads in engineering and scientific applications, such as earthquake detection [10–12], fluid mechanics and turbulence modeling [13,14], dynamical systems [15], and constitutive modeling [16,17]. A recent class of deep learning known as physics-informed neural networks (PINN) [18], where the network is trained simultaneously on both data and the governing differential equations, has been shown to be particularly well suited for solution and inversion of equations governing physical systems, in domains such as fluid mechanics [18,19], solid mechanics [20] and dynamical systems [21]. This increased interest in engineering and science is due to the increased availability of data and open-source platforms such as Theano [22], TensorFlow [23], MXNET [24], and Keras [25], which offer features such as high-performance computing and automatic differentiation [26].

Advances in deep learning have led to the emergence of different neural network architectures, including densely connected multi-layer deep neural networks (DNNs), convolutional neural networks (CNNs), recurrent neural

* Corresponding author.
E-mail address: ehsanh@mit.edu (E. Haghighat).

https://doi.org/10.1016/j.cma.2020.113552
0045-7825/

networks (RNNs) and residual networks (ResNets). This proliferation of network architectures, and the (often steep) learning curve for each package, makes it challenging for new researchers in the field to use deep learning tools in their computational workflows. In this paper, we introduce an open-source Python package, SciANN, developed on TensorFlow and Keras, which is designed with scientific computations and physics-informed deep learning in mind. As such, the abstractions used in this programming interface target engineering applications such as model fitting, solution of ordinary and partial differential equations, and model inversion (parameter identification).

The outline of the paper is as follows. We first describe the functional form associated with deep neural networks. We then discuss different interfaces in SciANN that can be used to set up neural networks and optimization problems. We then illustrate SciANN's application to curve fitting, the solution of the Burgers equation, and the identification of the Navier–Stokes equations and the von Mises plasticity model from data. Lastly, we show how to use SciANN in the context of the variational PINN framework [27]. The examples discussed here and several additional applications are freely available at github.com/sciann/sciann-applications.

2. Artificial neural networks as universal approximators

A single-layer feed-forward neural network with inputs $\mathbf{x} \in \mathbb{R}^m$, outputs $\mathbf{y} \in \mathbb{R}^n$, and d hidden units is constructed as:

$$\mathbf{y} = \mathbf{W}^1 \sigma(\mathbf{W}^0 \mathbf{x} + \mathbf{b}^0) + \mathbf{b}^1, \tag{1}$$

where ($\mathbf{W}^0 \in \mathbb{R}^{d\times m}$, $\mathbf{b}^0 \in \mathbb{R}^d$), ($\mathbf{W}^1 \in \mathbb{R}^{n\times d}$, $\mathbf{b}^1 \in \mathbb{R}^n$) are parameters of this transformation, also known as weights and biases, and σ is the activation function. As shown in [28,29], this transformation can approximate any measurable function, independently of the size of input features m or the activation function σ. If we define the transformation Σ as $\Sigma^i(\hat{\mathbf{x}}^i) := \hat{\mathbf{y}}^i = \sigma^i(\mathbf{W}^i \hat{\mathbf{x}}^i + \mathbf{b}^i)$ with $\hat{\mathbf{x}}^i$ as the input to and $\hat{\mathbf{y}}^i$ as the output of any hidden layer i, $\mathbf{x} = \hat{\mathbf{x}}^0$ as the main input to the network, and $\mathbf{y} = \Sigma^L(\hat{\mathbf{x}}^L)$ as the final output of the network, we can construct a general L-layer neural network as composition of Σ^i functions as:

$$\mathbf{y} = \Sigma^L \circ \Sigma^{L-1} \circ \cdots \circ \Sigma^0(\mathbf{x}), \tag{2}$$

with σ^i as activation functions that make the transformations nonlinear. Some common activation functions are:

$$\begin{aligned} &\text{ReLU} \quad : \hat{x} \mapsto \hat{x}^+, \\ &\text{sigmoid} : \hat{x} \mapsto 1/(1+e^{\hat{x}}), \\ &\tanh \quad : \hat{x} \mapsto (e^{\hat{x}} - e^{-\hat{x}})/(e^{\hat{x}} + e^{-\hat{x}}). \end{aligned} \tag{3}$$

In general, this multilayer feed-forward neural network is capable of approximating functions to any desired accuracy [28,30]. Inaccurate approximation may arise due to lack of a deterministic relation between input and outputs, insufficient number of hidden units, inadequate training, or poor choice of the optimization algorithm.

The parameters of the neural network, $\mathbf{W}^i$ and $\mathbf{b}^i$ of all layers $i = \{0 \dots L\}$, are identified through minimization using a back-propagation algorithm [31]. For instance, if we approximate a field variable such as temperature T with a multi-layer neural network as $T(\mathbf{x}) \approx \hat{T}(\mathbf{x}) = \mathcal{N}_T(\mathbf{x}; \mathbf{W}, \mathbf{b})$, we can set up the optimization problem as

$$\arg\min_{\mathbf{W},\mathbf{b}} \mathcal{L}(\mathbf{W}, \mathbf{b}) := \left\| T(\mathbf{x}^*) - \hat{T}(\mathbf{x}^*) \right\| = \left\| T(\mathbf{x}^*) - \mathcal{N}_T(\mathbf{x}^*; \mathbf{W}, \mathbf{b}) \right\|, \tag{4}$$

where $\mathbf{x}^*$ is the set of discrete training points, and $\|\circ\|_p$ is the mean squared norm. Note that one can use other choices for the loss function $\mathcal{L}$, such as mean absolute error or cross-entropy. The optimization problem (4) is nonconvex, which may require significant trial and error efforts to find an effective optimization algorithm and optimization parameters.

A key element contributing to the recent success of deep learning is associated with graph-based implementation of neural networks and automatic differentiation (AD). Every node on a graph represents a mathematical operation (e.g., addition or multiplication), and the derivatives of those operators can be traced using the chain rule, which results in secondary graphs. Unlike hand evaluation of derivatives, AD is safe from errors, and, unlike numerical finite-difference-type differentiation, its accuracy remains within machine precision. Although it was initially developed to facilitate optimization of complex neural networks, AD is particularly attractive in the context of PINN, since it can be readily used to evaluate derivatives of the solution variables with respect to independent variables or equation parameters, as needed for the solution of partial differential equations [26].

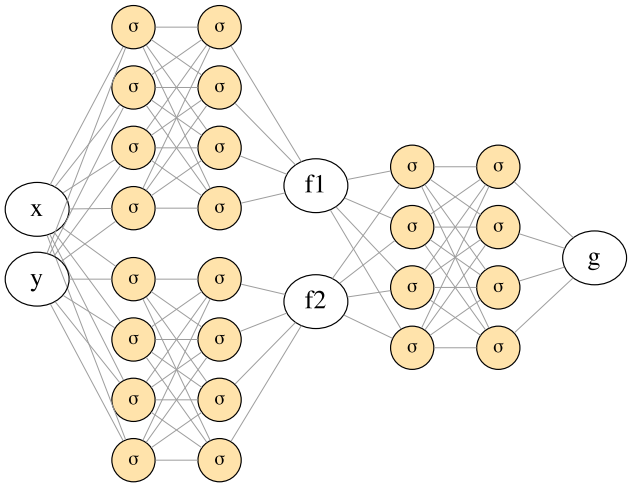

Fig. 1. A sample multi-net architecture to construct a complex functional space g as $g(x, y) = g(f_1(x, y), f_2(x, y))$.

We can construct deep neural networks with an arbitrary number of layers and neurons. We can also define multiple networks and combine them to generate the final output. There are many types of neural networks that have been optimized for specific tasks. An example is the ResNet architecture introduced for image classification, consisting of many blocks, each of the form:

$$\mathbf{z}^k = \Sigma^{k2} \circ \Sigma^{k1} \circ \Sigma^{k0}(\mathbf{z}^{k-1}) + \mathbf{z}^{k-1}, \tag{5}$$

where k is the block number and $\mathbf{z}^{k-1}$ is the output of previous block, with $\mathbf{x} = \mathbf{z}^0$ and $\mathbf{y} = \mathbf{z}^K$ as the main inputs to and outputs of the network. Therefore, artificial neural networks offer a simple way of constructing very complex but dependent solution spaces (see, e.g., Fig. 1).

3. SciANN: Scientific computing with artificial neural networks

SciANN is an open-source neural-network library, based on TensorFlow [23] and Keras [25], which abstracts the application of deep learning for scientific computing purposes. In this section, we discuss abstraction choices for SciANN and illustrate how one can use it for scientific computations.

3.1. Brief description of SciANN

SciANN is implemented on the most popular deep-learning packages, TensorFlow and Keras, and therefore it inherits all the functionalities they provide. Among those, the most important ones include graph-based automatic differentiation and massive heterogeneous high-performance computing capabilities. It is designed for an audience with a background in scientific computation or computational science and engineering.

SciANN currently supports fully connected feed-forward deep neural networks, and recurrent networks are under development. Some architectures, such as convolutional networks, are not a good fit for scientific computing applications and therefore are not currently in our development plans. TensorFlow and Keras provide a wide range of features, including optimization algorithms, automatic differentiation, and model parameter exports for transfer learning.

To install SciANN, one can simply use the Python's pip package installer as:

```
pip install sciann
```

It can be imported into the active Python environment using Python's import module:

```
import sciann as sn
```

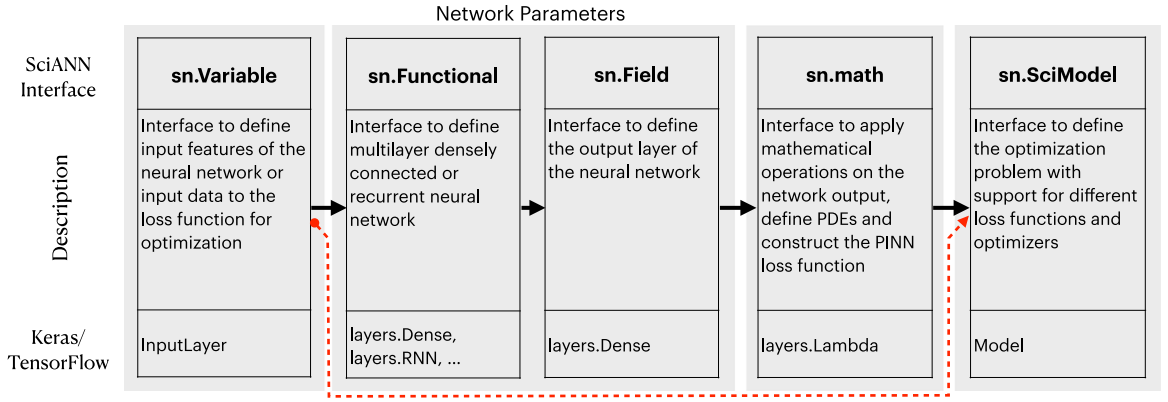

Fig. 2. The algorithmic abstraction used in SciANN and its connection to Keras/TensorFlow layers. `sn.Variable` handles all the inputs to the network or to the loss function. Operator overloading and different mathematical operations are handled using Keras's `Lambda` layers. The red dashed line highlights the use of `sn.Variable` for channeling data to the loss function.

Its mathematical functions are located in the `sn.math` interface. For instance, the function diff is accessed through `sn.math.diff`. The main building blocks of SciANN include:

- `sn.Variable`: class to define inputs to the network.
- `sn.Field`: class to define outputs of the network.
- `sn.Functional`: class to construct a nonlinear neural network approximation.
- `sn.Parameter`: class to define a parameter for inversion purposes.
- `sn.Data, sn.Tie`: class to define the targets. If there are observations for any variable, the 'sn.Data' interface is used when building the optimization model. For physical constraints such as PDEs or equality relations between different variables, the 'sn.Tie' interface is designed to build the optimizer.
- `sn.SciModel`: class to set up the optimization problem, i.e. inputs to the networks, targets (objectives), and the loss function.
- `sn.math`: mathematical operations are accessed here. SciANN also support operator overloading, which improves readability when setting up complex mathematical relations such as PDEs.

The algorithmic design of SciANN favors simplicity of use while taking advantage of key functionalities in Keras. Keras evaluates all terms of the loss function on a single training (collocation) grid through the input layer. This is done to achieve mini-batch optimization and parallelization capabilities. This is in contrast with the original PINN implementation [18], in which different terms of the loss function were evaluated on different grids; for example, the terms for the boundary condition and for the PDE are evaluated on different grids. Additionally, in SciANN, all mathematical operations on network outputs are handled through Keras's *Lambda* layers. Therefore, targets can be either a network output directly or the result of mathematical operations applied to network outputs. This choice makes the implementation of targets almost identical to their mathematical representations—a feature that will be illustrated in the following sections. In Fig. 2 we summarize the implementation flowchart for SciANN.

3.2. An illustrative example: curve fitting

We illustrate SciANN's capabilities with its application to a curve-fitting problem. Given a set of discrete data, generated from $f(x, y) = \sin(x)\sin(y)$ over the domain $x, y \to [-\pi, \pi] \times [-\pi, \pi]$, we want to fit a surface, in the form of a neural network, to this dataset. A multi-layer neural network approximating the function f can be constructed as $\hat{f} : (x, y) \mapsto \mathcal{N}_f(x, y; \mathbf{W}, \mathbf{b})$, with inputs x, y and output $\hat{f}$. In the most common mathematical and Pythonic abstraction, the inputs x, y and output $\hat{f}$ can be implemented as:

```
x = sn.Variable("x")
y = sn.Variable("y")
f = sn.Field("f")
```

A 3-layer neural network with 6 neural units and hyperbolic-tangent activation function can then be constructed as

```
f = sn.Functional(
    fields=[f],
    variables=[x, y],
    hidden_layers=[6, 6, 6],
    actf="tanh"
)
```

This definition can be further compressed as

```
f = sn.Functional("f", [x, y], [6, 6, 6], "tanh")
```

At this stage, the parameters of the networks, i.e. set of $\mathbf{W}$, $\mathbf{b}$ for all layers, are randomly initialized. Their current values can be retrieved using the command `get_weights`:

```
f.get_weights()
```

One can set the parameters of the network to any desired values using the command `set_weights`.

As another example, a more complex neural network functional as the composition of three blocks, as shown in Fig. 1, can be constructed as

```
f1 = sn.Functional("f1", [x, y], [4, 4], "tanh")
f2 = sn.Functional("f2", [x, y], [4, 4], "tanh")
g = sn.Functional("g", [f1, f2], [4, 4], "tanh")
```

Any of these functions can be evaluated immediately or after training using the `eval` function, by providing discrete data for the inputs:

```
f_test = f.eval([x_data, y_data])
f1_test = f1.eval([x_data, y_data])
f2_test = f2.eval([x_data, y_data])
g_test = g.eval([f1_data, f2_data])
```

Once the networks are initialized, we set up the optimization problem and *train* the network by minimizing an objective function, i.e. solving the optimization problem for $\mathbf{W}$ and $\mathbf{b}$. The optimization problem for a data-driven curve-fitting is defined as:

$$\arg\min_{\mathbf{W},\mathbf{b}} \mathcal{L}(\mathbf{W},\mathbf{b}) := \left\| f(x^*,y^*) - \mathcal{N}_f(x^*,y^*;\mathbf{W},\mathbf{b}) \right\|, \tag{6}$$

where x^*, y^* is the set of all discrete points where f is given. For the loss-function $\|\circ\|$, we use the mean squared-error norm $\|\circ\| = \frac{1}{N}\sum_{x^*,y^*\in I}(f(x^*,y^*) - \hat{f}(x^*,y^*))^2$. This problem is set up in SciANN through the `SciModel` class as:

```
m = sn.SciModel(
    inputs = [x, y],
    targets = [f],
    loss_func = "mse",
    optimizer = "adam"
)
```

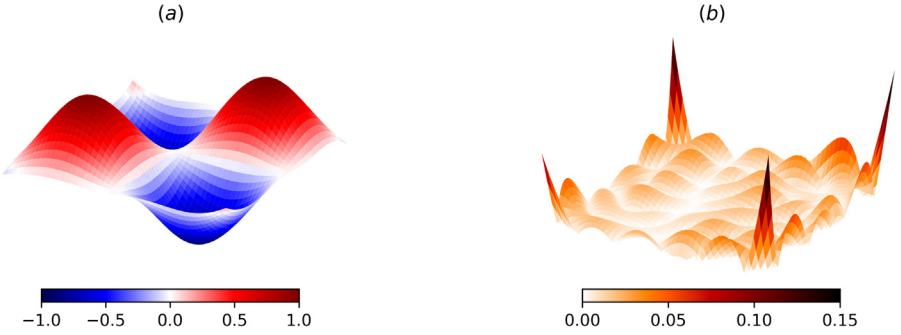

Fig. 3. Using SciANN to train a network on synthetic data generated from $\sin(x)\sin(y)$; (a): network predictions; (b): absolute error with respect to true values.

The `train` model is then used to perform the training and identify the parameters of the neural network:

```
m.train([x_data, y_data], [f_data], epochs=400)
```

Once the training is completed, one can set parameters of a `Functional` to be trainable or non-trainable (fixed). For instance, to set f to be non-trainable:

```
f1.set_trainable(False)
```

The result of this training is shown in Fig. 3, where we have used 400 epochs to perform the training on a dataset generated using a uniform grid of 51×51.

Since data was generated from $f(x, y) = \sin(x)\sin(y)$, we know that this is a solution to $\Delta f + 2f = 0$, with Δ as the Laplacian operator. As a first illustration of SciANN for physics-informed deep learning, we can constrain the curve-fitting problem with this 'governing equation'. In SciANN, the differentiation operators are evaluated through `sn.math.diff` function. Here, this differential equation can be evaluated as:

```
L = diff(fxy,x,order=2) + diff(fxy,y,order=2) + 2*fxy
```

with `order` expressing the order of differentiation.

Based on the physics-informed deep learning framework, the governing equation can be imposed through the objective function. The optimization problem can then be defined as

$$\arg\min_{\mathbf{W},\mathbf{b}} \mathcal{L}(\mathbf{W}, \mathbf{b}) := \left\| f(x^*, y^*) - \hat{f}(x^*, y^*) \right\| + \left\| \Delta\hat{f}(x^*, y^*) + 2\hat{f}(x^*, y^*) \right\|, \tag{7}$$

and implemented in SciANN as

```
m = SciModel([x, y], [fxy, L])
m.train([x_mesh, y_mesh], [(ids_data, fxy_data), 'zero'], epochs=400)
```

Note that while the inputs are the same as for the previous case, the optimization model is defined with two targets, `fxy` and `L`. The training data for `fxy` remains the same; the sampling grid, however, can be expanded further as 'physics' can be imposed everywhere. A sampling grid 101×101 is used here, where data is only given at the same locations as the previous case, i.e. on the 51×51 grid. To impose target `L`, it is simply set to `zero`. The new result is shown in Fig. 4. We find that, for the same network size and training parameters, incorporating the 'physics' reduces the error significantly.

Incorporating a new loss function is straightforward: one can include a new loss function when compiling the optimization problem using `sn.SciModel` through a Python function. Let us illustrate this process by the use of the

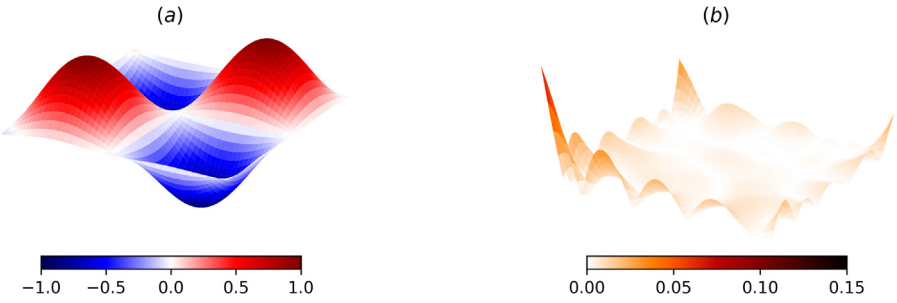

Fig. 4. Using SciANN to train a network on synthetic data generated from $\sin(x)\sin(y)$ and imposing the governing equations $f_{,xx} + f_{,yy} - 2f = 0$; (a): network predictions; (b): absolute error with respect to true values.

mean absolute error as the loss function. The *mean* operation is handled by default in the Keras backend, therefore one should define a function that evaluates the absolute error of each term and pass it to `sn.SciModel` using the `loss_func` keyword, as:

```
import keras.backend as K
def mean_abs_error(y_true, y_pred):
    return K.sum(K.abs(y_true - y_pred), axis=-1)
sn.SciModel(..., loss_func=mean_abs_error)
```

There are several factors that affect the training of a neural network, including the size of the network, the choice of activation function, the initial values of network parameters, and the optimization algorithm and its hyper parameters. If chosen identically, the trained network from SciANN is identical to that obtained when the network is implemented directly in TensorFlow/Keras. Once the training is completed, the weights $\mathbf{W}, \mathbf{b}$ for all layers can be saved using the command `save_weights`, for future use. These weights can be later used to initialize a network of the same structure using `load_weights_from` keyword in `SciModel`.

4. Application of SciANN to physics-informed deep learning

In this section, we explore how to use SciANN to solve and discover some representative case studies of physics-informed deep learning.

4.1. Burgers equation

As the first example, we illustrate the use of SciANN to solve the Burgers equation, which arises in fluid mechanics, acoustics, and traffic flow [32]. Following [33], we explore the governing equation:

$$u_{,t} + uu_{,x} - (0.01/\pi)u_{,xx} = 0, \quad t \in [0, 1], \quad x \in [-1, 1], \tag{8}$$

subject to initial and boundary conditions $u(t = 0, x) = -\sin(\pi x)$ and $u(t, x = \pm 1) = 0$, respectively. The solution variable u can be approximated by $\hat{u}$, defined in the form of a nonlinear neural network as $\hat{u} : (t, x) \mapsto \mathcal{N}_u(t, x; \mathbf{W}, \mathbf{b})$. The network used in [33] consists of 8 hidden layers, each with 20 neurons, and with tanh activation function, and can be defined in SciANN as:

```
t = sn.Variable("t")
x = sn.Variable("x")
u = sn.Functional("u", [t, x], 8*[20], "tanh")
```

To set up the optimization problem, we need to identify the targets. The first target, as used in the PINN framework, is the PDE in Eq. (8), and is defined in SciANN as:

```
import sciann.math.diff as diff
L1 = diff(u, t) + u*diff(u, x) - (0.01/pi)*diff(u, x, order=2)
```

To impose boundary conditions, one can define them as continuous mathematical functions defined at all sampling points:

$$
\begin{aligned}
L_2 &:= (1 - \text{sign}(t - t_{\text{min}}))(u + sin(\pi x)), \\
L_3 &:= (1 - \text{sign}(x - x_{\text{min}}))u, \\
L_4 &:= (1 + \text{sign}(x - x_{\text{max}}))u,
\end{aligned}
\tag{9}
$$

For instance, L_2 is zero at all sampling points except for $t < t_{\text{min}}$, which is chosen as $t_0 + \text{tol}$. Instead of sign, one can use smoother functions such as tanh.

```
import sciann.math.sign as sign
import sciann.math.sin as sin
L2 = (1 - sign(t - Tmin)) * (u + sin(pi*x))
L3 = (1 - sign(x - Xmin)) * u
L4 = (1 + sign(x - Xmax)) * u
```

In this way, the optimization model can be set up as:

```
m = sn.SciModel([t, x], [L1, L2, L3, L4], "mse", "Adam")
```

In this case, all targets should 'vanish', therefore the training is done as:

```
m.train(
    [x_data, t_data],
    ['zeros', 'zeros', 'zeros', 'zeros'],
    batch_size=256, epochs=10000
)
```

An alternative approach to define the boundary conditions in SciANN is to define the target in the `sn.SciModel` as the variable of interest and pass the 'ids' of training data where the conditions should be imposed. This is achieved as:

```
m = sn.SciModel([t, x], [L1, u], "mse", "Adam")
m.train(
    [x_data, t_data],
    ['zeros',  (ids_ic_bc, U_ic_bc)],
    batch_size=256, epochs=10000
)
```

Here, `ids_ic_bc` are ids associated with collocation points `(t_data, x_data)` where the initial condition and boundary condition are given. An important point to keep in mind is that if the number of sampling points where boundary conditions are imposed is a very small portion, the mini-batch optimization parameter `batch_size` should be set to a large number to guarantee consistent mini-batch optimization. Otherwise, some mini-batches may not acquire any data on the boundary and therefore not generate the correct gradient for the gradient-descent update. Also worth noting is that setting governing relations to 'zero' is conveniently done in SciANN.

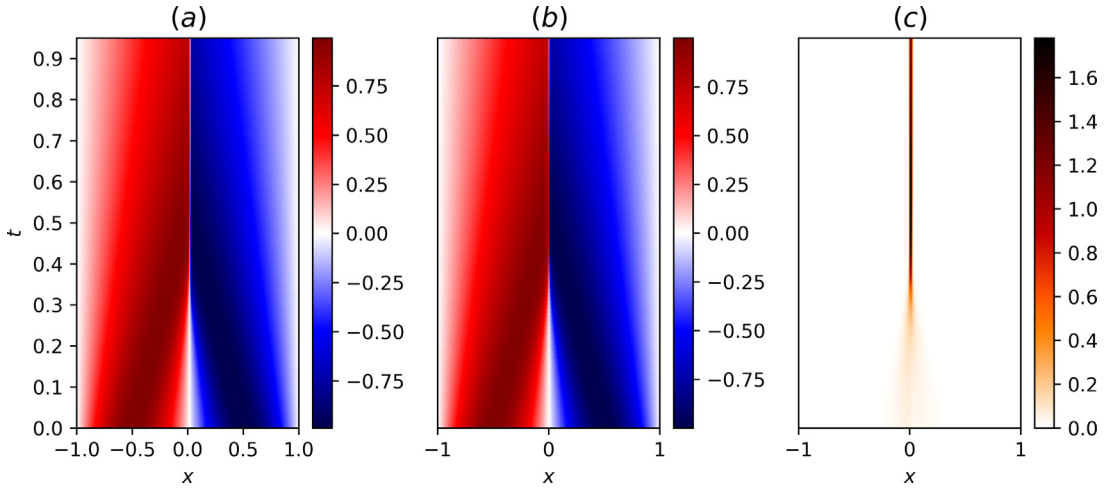

Fig. 5. Solution of the Burgers equation using PINN. (a) True solution for u; (b) PINN predicted values $\hat{u}$; (c) Absolute error between true and predicted values, $|u - \hat{u}|$.

The result of solving the Burgers equation using the deep learning framework is shown in Fig. 5. The results match the exact solution accurately, and reproduce the formation of a shock (self-sharpening discontinuity) in the solution at $x = 0$. The limited accuracy in the neighborhood of the boundary $x = 0$ is due to the coarse training grid; the accuracy can be improved by incorporating more collocation points in this region.

4.2. Data driven discovery of Navier–Stokes equations

As a second example, we show how SciANN can be used for discovery of partial differential equations. We choose the incompressible Navier–Stokes problem used in [18]. The equations are:

$$\begin{aligned} &u_{,t} + p_{,x} + \lambda_1(uu_{,x} + vu_{,y}) - \lambda_2(u_{,xx} + u_{,yy}) = 0, \\ &v_{,t} + p_{,y} + \lambda_1(uv_{,x} + vv_{,y}) - \lambda_2(v_{,xx} + v_{,yy}) = 0, \end{aligned} \tag{10}$$

where u and v are components of velocity field in x and y directions, respectively, p is the density-normalized pressure, λ_1 should be identically equal to 1 for Newtonian fluids, and λ_2 is the kinematic viscosity. The true value of the parameters to be identified are $\lambda_1 = 1$ and $\lambda_2 = 0.01$. Given the assumption of fluid incompressibility, we use the divergence-free form of the equations, from which the components of the velocity are obtained as:

$$u = \psi_{,y}, \quad v = -\psi_{,x}, \tag{11}$$

where ψ is the potential function.

Here, the independent field variables p and ψ are approximated as $p(t, x, y) \approx \hat{p}(t, x, y)$ and $\psi(t, x, y) \approx \hat{\psi}(t, x, y)$, respectively, using nonlinear artificial neural networks as $\hat{p} : (t, x, y) \mapsto \mathcal{N}_p(t, x, y; \mathbf{W}, \mathbf{b})$ and $\hat{\psi} : (t, x, y) \mapsto \mathcal{N}_\psi(t, x, y; \mathbf{W}, \mathbf{b})$. Using the same network size and activation function that was used in [18], we set up the neural networks in SciANN as:

```
p = sn.Functional("p", [t, x, y], 8*[20], 'tanh')
psi = sn.Functional("psi", [t, x, y], 8*[20], 'tanh')
```

Note that this way of defining the networks results in *two* separate networks for p and ψ, which we find more suitable for many problems. To replicate the one-network model used in the original study, one can use:

```
p, psi = sn.Functional(["p", "psi"], [t, x, y], 8*[20], 'tanh').split()
```

Here, the objective is to identify parameters λ_1 and λ_2 of the Navier–Stokes equations (10) on a dataset with given velocity field. Therefore, we need to define these as trainable parameters of the network. This is done using `sn.Parameter` interface as:

```
lamb1 = sn.Parameter(0.0, [x, y], name="lamb1")
lamb2 = sn.Parameter(0.0, [x, y], name="lamb2")
```

Note that these parameters are initialized with a value of 0.0. The required derivatives in Eqs. (10) and (11) are evaluated as:

```
u, v = diff(psi,y), -diff(psi,x)
u_t, v_t = diff(u,t), diff(v,t)
u_x, u_y = diff(u,x), diff(u,y)
v_x, v_y = diff(v,x), diff(v,y)
u_xx, u_yy = diff(u,x,order=2), diff(u,y,order=2)
v_xx, v_yy = diff(v,x,order=2), diff(v,y,order=2)
p_x, p_y = diff(p,x), diff(p,y)
```

with 'order' indicating the order of differentiation. We can now set up the targets of the problem as:

```
L1 = u_t + p_x + lamb1*(u*u_x + v*u_v) - lamb2*(u_xx + u_yy)
L2 = v_t + p_y + lamb1*(u*v_x + v*v_y) - lamb2*(v_xx + v_yy)
L3 = u
L4 = v
```

The optimization model is now set up as:

```
m = sn.SciModel([t, x, y], [L1, L2, L3, L4], "mse", "Adam")
m.train([t_data, x_data, y_data],
        ['zeros', 'zeros', u_data, v_data],
        batch_size=64, epochs=10000)
```

where only training points for u and v are provided, as in [18]. The results are shown in Fig. 6, that are the same as those reported in [18].

4.3. Discovery of nonlinear solid mechanics with von Mises plasticity

Here, we illustrate the use of PINN for solution and discovery of nonlinear solid mechanics. We use the von Mises elastoplastic constitutive model, which is commonly used to describe mechanical behavior of solid materials, in particular metals. Elastoplasticity relations give rise to inequality constraints on the governing equations [34], and, therefore, compared to the Navier–Stokes equations, they pose a different challenge to be incorporated in PINN. The elastoplastic relations for a plane-strain problem are:

$$\begin{aligned}
\sigma_{ij,j} + f_i &= 0,\\
\sigma_{ij} &= s_{ij} - p\delta_{ij},\\
p &= -\sigma_{kk}/3 = -(\lambda + 2/3\mu)\varepsilon_v,\\
s_{ij} &= 2\mu e^e_{ij},\\
\varepsilon_{ij} &= (u_{i,j} + u_{j,i})/2 = e_{ij} + \varepsilon_v\delta_{ij}/3,\\
\varepsilon_v &= \varepsilon_{kk} = \varepsilon_{xx} + \varepsilon_{yy},\\
e_{ij} &= e^e_{ij} + e^p_{ij}.
\end{aligned} \tag{12}$$

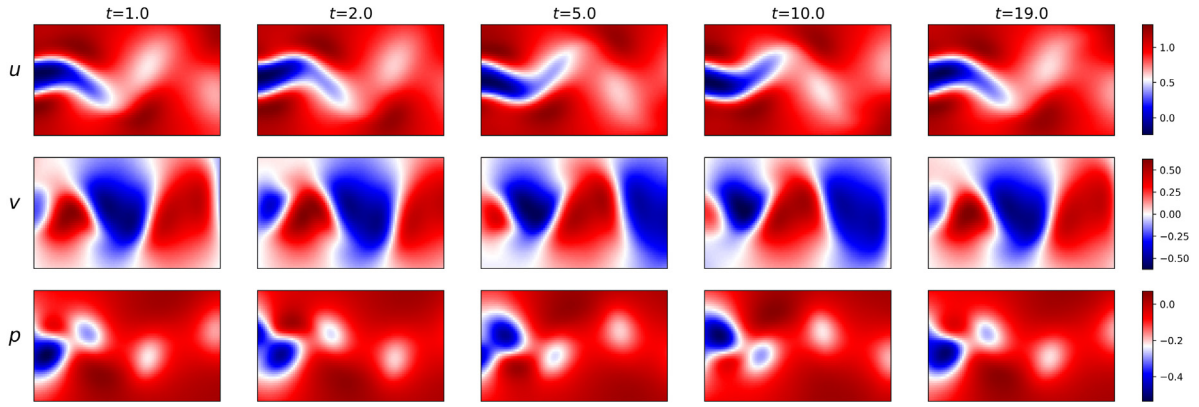

Fig. 6. Predicted values from the PINN framework, for the field variables u, v and p, at different times t. The parameters are identified as $\lambda_1 = 0.9967$ and $\lambda_2 = 0.0110$. The predictions are the same as those reported in [18].

Here, the summation notation is used with $i, j, k \in \{x, y\}$. σ_{ij} are components of the Cauchy stress tensor, and s_{ij} and p are its deviatoric components and its pressure invariant, respectively. ε_{ij} are components of the infinitesimal strain tensor derived from the displacements u_x, u_y, and e_{ij} and ε_v are its deviatoric and volumetric components, respectively.

According to the von Mises plasticity model, the admissible state of stress is defined inside the cylindrical yield surface $\mathcal{F} = \mathcal{F}(\sigma_{ij})$ as $\mathcal{F} := q - \sigma_Y \leq 0$. Here, q is the equivalent stress defined as $q = \sqrt{3/2 s_{ij} s_{ij}}$. Assuming the associative flow rule, the plastic strain components are:

$$\varepsilon^p_{ij} \equiv e^p_{ij} = \bar{e}^p \frac{\partial \mathcal{F}}{\partial \sigma_{ij}} = \bar{\varepsilon}^p \frac{3}{2} \frac{s_{ij}}{q}, \tag{13}$$

where $\bar{\varepsilon}^p$ is the equivalent plastic strain, subject to $\bar{\varepsilon}^p \geq 0$. For the von Mises model, it can be shown that $\bar{\varepsilon}^p$ is evaluated as

$$\bar{\varepsilon}^p = \bar{\varepsilon} - \frac{\sigma_Y}{3\mu} \geq 0, \tag{14}$$

where $\bar{\varepsilon}$ is the total equivalent strain, defined as $\bar{\varepsilon} = \sqrt{2/3 e_{ij} e_{ij}}$. Note that for von Mises plasticity, the volumetric part of plastic strain tensor is zero, $\varepsilon^p_v = 0$. Finally, the parameters of this model include the Lamé elastic parameters λ and μ, and the yield stress σ_Y.

We use a classic example to illustrate our framework: a perforated strip subjected to uniaxial extension [34,35]. Consider a plate of dimensions 200 mm × 360 mm, with a circular hole of diameter 100 mm located in the center of the plate. The plate is subjected to extension displacements of $\delta = 1$ mm along the short edge, under plane-strain condition, and without body forces, $f_i = 0$. The parameters are $\lambda = 19.44$ GPa, $\mu = 29.17$ GPa and $\sigma_Y = 243.0$ MPa. Due to symmetry, only a quarter of the domain needs to be considered in the simulation. The synthetic data is generated from a high-fidelity FEM simulation using COMSOL software [36] on a mesh of approximately 13,000 quartic triangular elements. The plate undergoes significant plastic deformation around the circular hole. This results in localized deformation in the form of a shear band. While the strain exhibits localization, the stress field remains continuous and smooth—a behavior that is due to the choice of a perfect-plasticity model with no hardening.

Following the approach proposed in [20], we approximate displacement and stress components $u_x, u_y, \sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}$ with nonlinear neural networks as:

$$\begin{aligned}
\hat{u}_x &: (x, y) \mapsto \mathcal{N}_{u_x}(x, y; \mathbf{W}, \mathbf{b}) \\
\hat{v}_x &: (x, y) \mapsto \mathcal{N}_{v_x}(x, y; \mathbf{W}, \mathbf{b}) \\
\hat{\sigma}_{xx} &: (x, y) \mapsto \mathcal{N}_{\sigma_{xx}}(x, y; \mathbf{W}, \mathbf{b}) \\
\hat{\sigma}_{yy} &: (x, y) \mapsto \mathcal{N}_{\sigma_{yy}}(x, y; \mathbf{W}, \mathbf{b}) \\
\hat{\sigma}_{zz} &: (x, y) \mapsto \mathcal{N}_{\sigma_{zz}}(x, y; \mathbf{W}, \mathbf{b}) \\
\hat{\sigma}_{xy} &: (x, y) \mapsto \mathcal{N}_{\sigma_{xy}}(x, y; \mathbf{W}, \mathbf{b})
\end{aligned} \tag{15}$$

Note that due to plastic deformation, the out-of-plane stress σ_{zz} is not predefined, and therefore we also approximate it with a neural network. These neural networks and parameters λ, μ, σ_Y are defined as follows:

```
ux = sn.Functional('ux', [x, y], 4*[50], 'tanh')
uy = sn.Functional('uy', [x, y], 4*[50], 'tanh')
sxx = sn.Functional('sxx', [x, y], 4*[50], 'tanh')
syy = sn.Functional('syy', [x, y], 4*[50], 'tanh')
szz = sn.Functional('szz', [x, y], 4*[50], 'tanh')
sxy = sn.Functional('sxy', [x, y], 4*[50], 'tanh')
lmbd = sn.Paramater(1.0, [x, y])
mu = sn.Paramater(1.0, [x, y])
sy = sn.Paramater(1.0, [x, y])
```

The kinematic relations, deviatoric stress components and plastic strains can be defined as:

```
# Total strain components
Exx = diff(ux, x)
Eyy = diff(uy, y)
Exy = (diff(ux, y) + diff(uy, x))/2
Evol = Exx + Eyy
# Deviatoric strain components
exx = Exx - Evol/3
eyy = Eyy - Evol/3
ezz = - Evol/3 # Ezz=0 (plane strain)
ebar = sn.math.sqrt(2/3*(exx**2 + eyy**2 + ezz**2 + 2*exy**2))
# Deviatoric stress components
prs = -(sxx + syy + szz)/3
dxx = sxx + prs
dyy = syy + prs
dzz = szz + prs
q = sn.math.sqrt(3/2*(dxx**2 + dyy**2 + dzz**2 + 2*sxy**2))
# Plastic strain components
pebar = sn.math.relu(ebar - sy/(3*mu))
pexx = 1.5 * pebar * sxx / q
peyy = 1.5 * pebar * syy / q
pezz = 1.5 * pebar * szz / q
pexy = 1.5 * pebar * sxy / q
# Yield surface
F = q - sy
```

The operator-overloading abstraction of SciANN improves readability significantly. Assuming access to the measured data for variables u_x, u_y, σ_{xx}, σ_{yy}, σ_{zz}, σ_{xy}, ε_{xx}, ε_{yy}, ε_{xy}, the optimization targets for training data can be described using the `L* = sn.Data(*)`, where $*$ refers to each variable. The physics-informed constraints are set as:

```
# Volumetric (hydro-static) stress
L1 = sn.Tie(prs, -kappa*Evol)
# Stress relations
L2 = sn.Tie(dxx, 2*mu*(exx - pexx))
L3 = sn.Tie(dyy, 2*mu*(eyy - peyy))
L4 = sn.Tie(dzz, 2*mu*(ezz - pezz))
L5 = sn.Tie(dxy, 2*mu*(exy - pexy))
# Yield surface
# Penalize the positive part
L6 = sn.math.relu(F)
# Momentum relations
L7 = sn.diff(sxx, x) + sn.diff(sxy, y)
L8 = sn.diff(sxy, x) + sn.diff(syy, y)
```

We use 2000 data points from this reference solution, randomly distributed in the simulation domain, to provide the training data. The PINN training is performed using networks with 4 layers, each with 100 neurons, and with a hyperbolic-tangent activation function. The optimization parameters are the same as those used in [20]. The results predicted by the PINN approach match the reference results very closely, as evidenced by: (1) the very small errors in each of the components of the solution, except for the out-of-plane plastic strain components (Fig. 7); and (2) the precise identification of yield stress σ_Y and relatively accurate identification of elastic parameters λ and μ, yielding estimated values $\lambda = 18.3$ GPa, $\mu = 27.6$ GPa and $\sigma_Y = 243.0$ MPa.

5. Application to variational PINN

Neural networks have recently been used to solve the *variational form* of differential equations as well [37,38]. In a recent study [27], the vPINN framework for solving PDEs was introduced and analyzed. Like PINN, it is based on graph-based automatic differentiation. The authors of [27] suggest a Petrov–Galerkin approach, where the test functions are chosen differently from the trial functions. For the test functions, they propose the use of polynomials that vanish on the boundary of the domain. Here, we illustrate how to use SciANN for vPINN, and we show how to construct proper test functions using neural networks.

Consider the steady-state heat equation subject to Dirichlet boundary conditions and a known heat source $f(x, y)$ [27]:

$$\Delta T + f(x, y) = 0, \quad x, y \in [-1, 1] \times [-1, 1], \tag{16}$$

subject to the following boundary conditions:

$$\begin{aligned} &T(x = \pm 1, y) = \sin(2\pi y), \\ &T(x, y = \pm 1) = 0, \end{aligned} \tag{17}$$

and a heat source:

$$\begin{aligned} f(x, y) = \sin(2\pi y) &\left(20 \tanh(10x) \left(10 \tanh(10x)^2 - 10 \right) - \frac{2\pi^2 \sin(2\pi x)}{5} \right) \\ &- 4\pi^2 \sin(2\pi y) \left(\tanh(10x) + \frac{\sin(2\pi x)}{10} \right). \end{aligned} \tag{18}$$

The analytical solution to this problem is:

$$T(x, y) = (0.1 \sin(2\pi x) + \tanh(10x)) \sin(2\pi y). \tag{19}$$

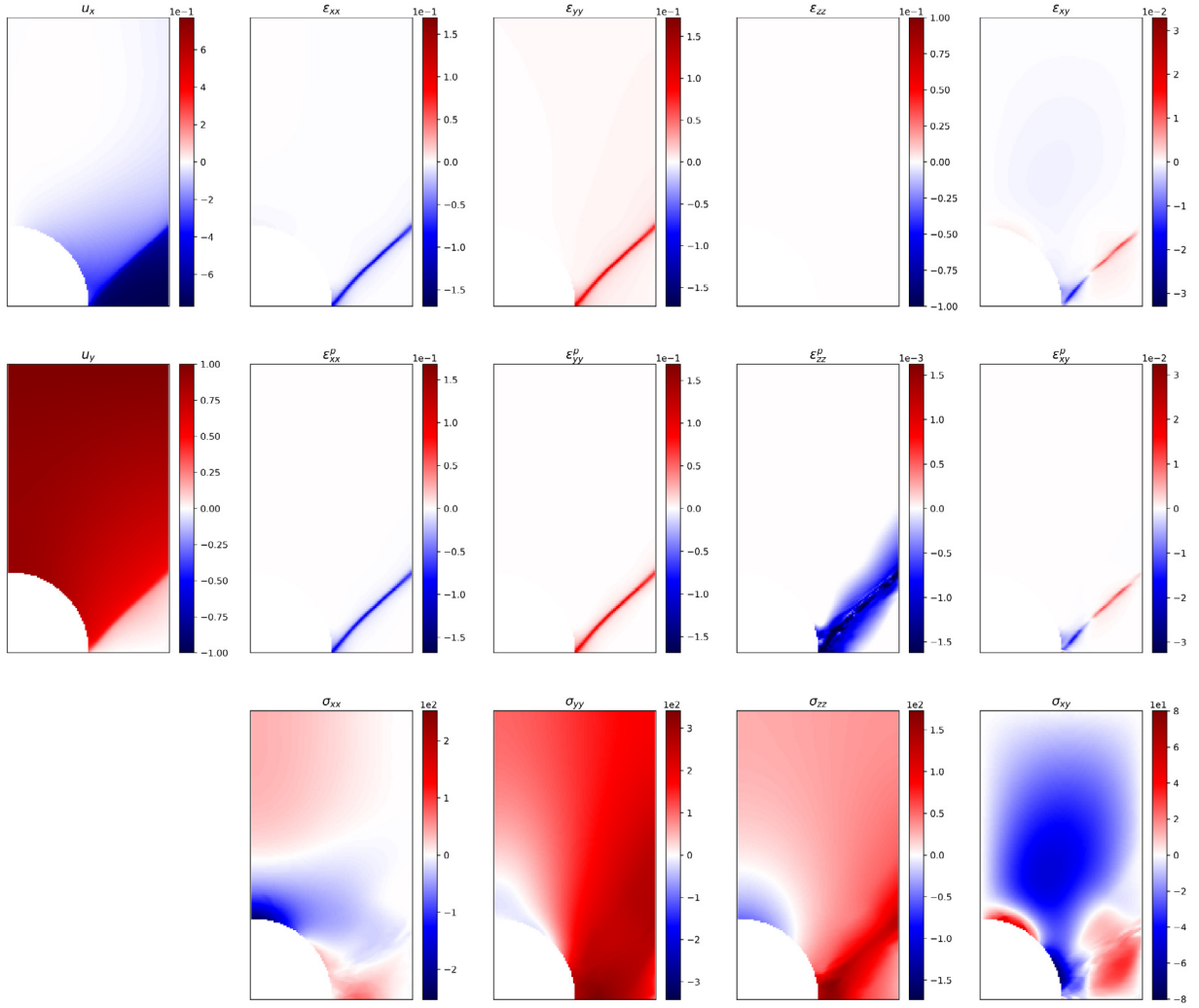

Fig. 7. The predicted values from the PINN framework for displacements, strains, plastic strains and stresses. The inverted parameters are $\lambda = 18.3$ GPa, $\mu = 27.6$ GPa and $\sigma_Y = 243.0$ MPa.

The weak form of Eq. (16) is expressed as:

$$\int_{\Omega} \left[\nabla Q \cdot \nabla T + Q \cdot f(x, y)\right] dV = \int_{\partial\Omega} Q q_n \, dS, \tag{20}$$

where Ω is the domain of the problem, $\partial\Omega$ is the boundary of the domain, q_n is the boundary heat flux, and Q is the test function. The trial space for the temperature field T is constructed by a neural network as $T : (x, y) \mapsto \mathcal{N}_T(x, y; \mathbf{W}, \mathbf{b})$. For the test space Q, the authors of [27] suggest the use of polynomials that satisfy the boundary conditions. However, considering the universal approximation capabilities of the neural networks, we suggest that this step is unnecessary, and a general neural network can be used as the test function. Note that test functions should satisfy continuity requirements as well as boundary conditions. A multi-layer neural network with any nonlinear activation function is a good candidate for the continuity requirements. To satisfy the boundary conditions, we can simply train the test functions to vanish on the boundary. Note that this step is associated to the construction of proper test function and is done as a preprocessing step. Once the test functions satisfy the (homogeneous) boundary conditions, there is no need to further train them, and therefore their parameters can be set to non-trainable at this stage. We also find that there is no need for the $\mathcal{N}_T$ and $\mathcal{N}_Q$ networks to be of the same size, or use the same activation functions.

Therefore, the test function Q is defined as $Q : (x, y) \mapsto \mathcal{N}_Q(x, y; \bar{\mathbf{W}}, \bar{\mathbf{b}})$ subject to $Q(x = \pm 1, y) = Q(x, y \pm = 1) = 0$. Here, overbar weights and biases $\bar{\mathbf{W}}, \bar{\mathbf{b}}$ indicate that their values are predefined and fixed (non-trainable). Therefore, the boundary flux integral on the right side of Eq. (20) vanishes, and the resulting week form can be expressed as

$$\int_{\Omega} \nabla Q \cdot \nabla T + Q \cdot f(x, y)\, dV = 0. \tag{21}$$

The problem can be defined in SciANN as follows. The first step is to construct proper test function:

```
Q = sn.Functional('Q', [x, y], 4*[20], 'sigmoid')
m = sn.SciModel([x, y], [Q])
m.train([x_data, y_data], [Q_data])
Q.set_trainable(False)
```

As discussed earlier, `Q_data` takes a value of 0.0 for training points on the boundary and random values at interior quadrature points. Additionally, parameters of Q are set to non-trainable at the end of this step. The trial function T and the target weak form in Eq. (21) are now implemented as:

```
T = sn.Functional('T', [x, y], 4*[20], 'tanh')
Q_x, Q_y = diff(Q, x), diff(Q, y)
T_x, T_y = diff(T, x), diff(T, y)
# New variables for body force and volume information
fxy = sn.Variable('fxy')
vol = sn.Variable('vol')
# The variational target
J = (Q_x*T_x + Q_y*T_y + Q*fxy) * vol
```

Since the variational relation (21) takes an integral form, we need to perform a domain integral. Therefore, the volume information should be passed to the network along with the body-force information at the quadrature points. This is achieved by introducing two new SciANN variables as the inputs to the network. The optimization model is then defined as:

```
m = sn.SciModel([x, y, vol], [J, T], "mse")
m.train(
    [x_data, y_data, vol_data, fxy_data],
    ['zeros', (bc_ids, bc_vals)],
)
```

The second target on T imposes the boundary conditions at specific quadrature points `bc_ids`.

Following the details in [27], we perform the integration on a 70×70 grid. The results are shown in Fig. 8, which are very similar to those reported in [27].

6. Conclusions

In this paper, we have introduced the open-source deep-learning package, SciANN, designed specifically to facilitate physics-informed simulation, inversion, and discovery in the context of computational science and engineering problems. It can be used for regression and physics-informed deep learning with minimal effort on the neural network setup. It is based on TensorFlow and Keras packages, and therefore it inherits all the high-performance computing capabilities of TensorFlow back-end, including CPU/GPU parallelization capabilities.

The objective of this paper is to introduce an environment based on a modern implementation of graph-based neural network and automatic differentiation, to be used as a platform for scientific computations. In a series of examples, we have shown how to use SciANN for curve-fitting, solving PDEs in strong and weak form, and for

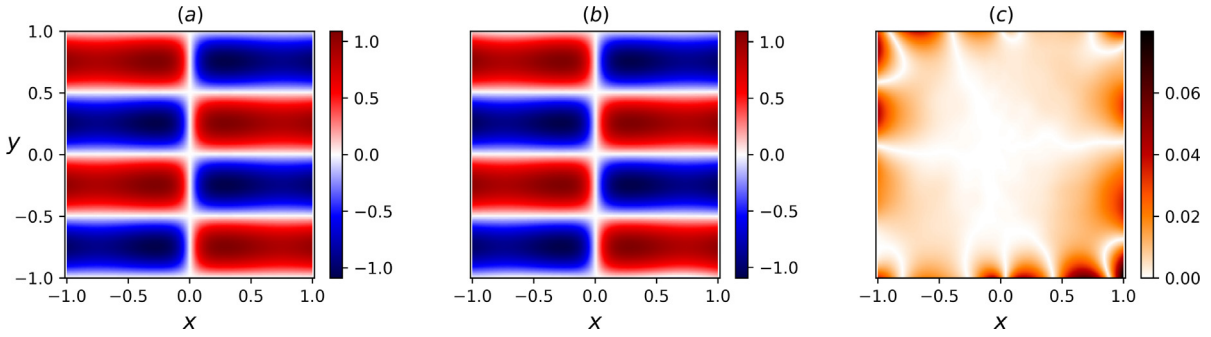

Fig. 8. Solution of a steady-state heat equation using the vPINN framework. (a) True temperature field, $T(x, y)$. (b) Temperature field predicted by the neural network, $\hat{T}(x, y)$. (c) Absolute error between true and predicted values, $|T(x, y) - \hat{T}(x, y)|$.

model inversion in the context of physics-informed deep learning. As part of the continued development of SciANN, we are currently working on the implementation of several attractive features for scientific computing, including domain decomposition, stochastic modeling, and uncertainty quantification. The examples presented here as well as the package itself are all open-source, and available in the github repository github.com/sciann.

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.

Acknowledgments

This work was funded by the KFUPM-MIT, United States, collaborative agreement 'Multiscale Reservoir Science'.

References

[1] C.M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006, URL http://www.springer.com/us/book/9780387310732.
[2] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in: F. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger (Eds.), Advances in Neural Information Processing Systems 25, MIT Press, 2012, pp. 1097–1105, URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.
[3] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444, http://dx.doi.org/10.1038/nature14539.
[4] D. Jannach, M. Zanker, A. Felfernig, G. Friedrich, Recommender Systems: An Introduction, Cambridge University Press, 2010.
[5] S. Zhang, L. Yao, A. Sun, Y. Tay, Deep learning based recommender system: A survey and new perspectives, ACM Comput. Surv. 52 (1) (2019) 1–38.
[6] A. Graves, A.-R. Mohamed, G. Hinton, Speech recognition with deep recurrent neural networks, in: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, IEEE, 2013, pp. 6645–6649.
[7] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L.D. Jackel, M. Monfort, U. Muller, J. Zhang, et al., End to end learning for self-driving cars, 2016, arXiv preprint arXiv:1604.07316.
[8] R. Miotto, F. Wang, S. Wang, X. Jiang, J.T. Dudley, Deep learning for healthcare: review, opportunities and challenges, Brief. Bioinform. 19 (6) (2018) 1236–1246.
[9] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016, URL https://www.deeplearningbook.org.
[10] Q. Kong, D.T. Trugman, Z.E. Ross, M.J. Bianco, B.J. Meade, P. Gerstoft, Machine learning in seismology: turning data into insights, Seismol. Res. Lett. 90 (1) (2018) 3–14, http://dx.doi.org/10.1785/0220180259.
[11] Z.E. Ross, D.T. Trugman, E. Hauksson, P.M. Shearer, Searching for hidden earthquakes in Southern California, Science 364 (6442) (2019) 767–771, http://dx.doi.org/10.1126/science.aaw6888.
[12] K.J. Bergen, P.A. Johnson, M.V. de Hoop, G.C. Beroza, Machine learning for data-driven discovery in solid earth geoscience, Science 363 (6433) (2019) eaau0323, http://dx.doi.org/10.1126/science.aau0323.
[13] M.P. Brenner, J.D. Eldredge, J.B. Freund, Perspective on machine learning for advancing fluid mechanics, Phys. Rev. Fluids 4 (10) (2019) 100501, http://dx.doi.org/10.1103/PhysRevFluids.4.100501.
[14] S.L. Brunton, B.R. Noack, P. Koumoutsakos, Machine learning for fluid mechanics, Annu. Rev. Fluid Mech. 52 (1) (2020) 477–508, http://dx.doi.org/10.1146/annurev-fluid-010719-060214.
[15] S. Dana, M.F. Wheeler, A machine learning accelerated FE^2 homogenization algorithm for elastic solids, 2020, arXiv:2003.11372.

[16] A.M. Tartakovsky, C.O. Marrero, P. Perdikaris, G.D. Tartakovsky, D. Barajas-Solano, Learning parameters and constitutive relationships with physics informed deep neural networks, 2018, arXiv:1808.03398.
[17] K. Xu, D.Z. Huang, E. Darve, Learning constitutive relations using symmetric positive definite neural networks, 2020, pp. 1–31, arXiv:2004.00265..
[18] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, J. Comput. Phys. 378 (2019) 686–707, http://dx.doi.org/10.1016/j.jcp.2018.10.045.
[19] M. Raissi, A. Yazdani, G.E. Karniadakis, Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations, Science 367 (6481) (2020) 1026–1030, http://dx.doi.org/10.1126/science.aaw4741.
[20] E. Haghighat, M. Raissi, A. Moure, H. Gomez, R. Juanes, A deep learning framework for solution and discovery in solid mechanics, 2020, arXiv:2003.02751..
[21] S. Rudy, A. Alla, S.L. Brunton, J.N. Kutz, Data-driven identification of parametric partial differential equations, SIAM J. Appl. Dyn. Syst. 18 (2) (2019) 643–660, http://dx.doi.org/10.1137/18M1191944.
[22] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, Y. Bengio, Theano: a CPU and GPU math expression compiler, in: Proceedings of the Python for Scientific Computing Conference (SciPy), Vol. 4, Austin, TX, 2010.
[23] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, Tensorflow: A system for large-scale machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association, Savannah, GA, 2016, pp. 265–283, URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.
[24] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang, MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015, arXiv:1512.01274.
[25] F. Chollet, Deep Learning with Python, Manning Publications Company, 2017, URL https://books.google.ca/books?id=Yo3CAQAACAAJ.
[26] A. Güne, G. Baydin, B.A. Pearlmutter, J.M. Siskind, Automatic differentiation in machine learning: a survey, J. Mach. Learn. Res. 18 (2018) 1–43, URL http://www.jmlr.org/papers/volume18/17-468/17-468.pdf.
[27] E. Kharazmi, Z. Zhang, G.E. Karniadakis, Variational physics-informed neural networks for solving partial differential equations, 2019, pp. 1–24, arXiv:1912.00873..
[28] K. Hornik, M. Stinchcombe, H. White, Multilayer feed-forward networks are universal approximators, Neural Netw. 2 (5) (1989) 359–366, http://dx.doi.org/10.1016/0893-6080(89)90020-8.
[29] G. Cybenko, Approximation by superpositions of a sigmoidal function, Math. Control Signals Systems 2 (4) (1989) 303–314, http://dx.doi.org/10.1007/BF02551274.
[30] K. Hornik, Approximation capabilities of multilayer feed-forward networks, Neural Netw. 4 (2) (1991) 251–257, http://dx.doi.org/10.1016/0893-6080(91)90009-T.
[31] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning representations by back-propagating errors, Nature 323 (6088) (1986) 533–536, http://dx.doi.org/10.1038/323533a0.
[32] C.M. Dafermos, Hyperbolic Conservation Laws in Continuum Physics, Springer-Verlag, Berlin, 2000.
[33] M. Raissi, Deep hidden physics models: Deep learning of nonlinear partial differential equations, J. Mach. Learn. Res. 19 (2018) 1–24, arXiv:1801.06637.
[34] J.C. Simo, T.J.R. Hughes, Computational Inelasticity, in: Interdisciplinary Applied Mathematics, vol. 7, Springer, New York, 1998.
[35] O. Zienkiewicz, S. Valliappan, I. King, Elasto-plastic solutions of engineering problems 'initial stress', finite element approach, Internat. J. Numer. Methods Engrg. 1 (1) (1969) 75–100.
[36] COMSOL, COMSOL Multiphysics User's Guide, COMSOL, Stockholm, Sweden, 2020.
[37] E. Weinan, B. Yu, The deep Ritz method: A deep learning-based numerical algorithm for solving variational problems, Commun. Math. Stat. 6 (1) (2018) 1–14, http://dx.doi.org/10.1007/s40304-018-0127-z, arXiv:1710.00211.
[38] J. Berg, K. Nyström, A unified deep artificial neural network approach to partial differential equations in complex geometries, Neurocomputing 317 (2018) 28–41, http://dx.doi.org/10.1016/j.neucom.2018.06.056, arXiv:1711.06464.