Supplementary material - Solving differential equations in physics with Deep Learning: a beginner's guide

Luis Medrano Navarro

Instituto de Nanociencia y Materiales de Aragón (INMA), CSIC-Universidad de Zaragoza, 50009 Zaragoza, Spain

E-mail: 780070@unizar.es; ORCID=0000-0003-3246-9383

Luis Martin Moreno

Instituto de Nanociencia y Materiales de Aragón (INMA), CSIC-Universidad de Zaragoza, 50009 Zaragoza, Spain
Departamento de Física de la Materia Condensada, Universidad de Zaragoza, Zaragoza 50009, Spain

E-mail: lmm@unizar.es; ORCID=0000-0001-9273-8165

Sergio G Rodrigo

Departamento de Física Aplicada, Facultad de Ciencias, Universidad de Zaragoza, 50009 Zaragoza, Spain
Instituto de Nanociencia y Materiales de Aragón (INMA), CSIC-Universidad de Zaragoza, 50009 Zaragoza, Spain

E-mail: sergut@unizar.es; ORCID=0000-0001-6575-168X

1. Deep learning PINNs with Tensorflow-Keras

In the following, we describe the *Python* codes that were implemented to conduct the calculations of the paper. These codes can be found in the following Github repository [1]. We provide a comprehensive explanation of them, such that it can be utilized effectively and modified according to one's requirements. The Supplementary Material of this paper also includes three *Jupyter* notebooks, so the reader can practice with the examples provided and build its own PINNs.

To implement our PINNs, we have used Keras and Tensorflow libraries [**?**]. Tensorflow consists of a set of programming libraries to operate with tensors. With Tensorflow, it is possible to implement neural networks (NN) from scratch. However, being a low-level library, its learning and use are relatively complex. On the other hand, Keras is a high-level Application Programming Interface (API) where it is easier to create complex architectures with NN. We have used Keras as the backbone of the implementation, but it has been necessary to use Tensorflow to generate the Ordinary Differential Equation (ODE) loss functions described in the paper.

2. Example 1: 1st order Ordinary Differential Equations

The first-order ODE to be solved with the use of PINNs is:

$$\begin{cases} y'(x) + y(x) = 0 \quad with \quad 0 < x < 4 \\ y(0) = 1 \end{cases} \tag{1}$$

, whose exact solution is $y(x) = \exp(-x)$.

2.1. Main libraries

To begin with, we start the loading of essential packages for the algorithm. Primarily, we load Tensorflow, and in addition, we utilize Numpy for mathematical and array operations, and Matplotlib for generating plots. We further import from Keras two types of layers (*Input* and *Dense*) and the *Adam* optimizer.

```
# Tensorflow Keras and the rest of the packages
import tensorflow as tf
from tensorflow.keras.layers import Input,Dense
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt
```

2.2. Definition of the PINN

The next lines of code can be considered the core of the PINN algorithm. In there we create the loss function in Tensorflow using the differential equation information.

The Python code defines a custom Keras model class *ODE_1st* that inherits from the *tf.keras.Model* class. The *train_step* method of this class implements the training loop for the model (see Ref. [3], for an introductory description of custom objects using Tensorflow-Keras). Inside the *train_step* method, the *data* argument is a tuple that contains the training inputs x and the analytical (exact) solution *y_exact* at these input points.

The word $self$ in this code means *model*. So, for example, $self(x0, training = True)$ calculates the model's prediction at the point of the initial condition x_0. This is why the result is *y0_NN* in the corresponding line of code, the prediction of the NN at this point.

Let's describe the different parts of the Python code:

(i) The method starts by defining the initial conditions for the PINN, as *x0* and *y0_exact*, which are set as constant tensors. It is mandatory that all variables defined throughout the code are in the Tensorflow format.

(ii) The code then computes the gradients of the output *y_NN* with respect to the input x and the initial condition *y0_NN*, using two nested *tf.GradientTape* contexts. The *tf.GradientTape* is the part of Tensorflow dedicated to Automatic Differentiation (AD). This is a very efficient way of calculating derivatives using NN, allowing the NN to solve differential equations. In particular, Tensorflow uses what is called reverse AD, based on the mathematical properties of dual numbers [3].

(iii) The loss function is then calculated using the computed gradients, and the initial conditions are included in the loss calculation. The loss is a sum of two contributions of $self.compiled\_loss$, which in our

case is the Mean Squared Error (MSE) function, described in the article. The line $self.compiled\_loss(dy\_dx\_NN, -y\_NN)$ returns the error in the differential equation ($|y' + y|^2$). The code line calling to $self.compiled\_loss(y0\_NN, y0\_exact)$ is the error in the initial condition ($|y(0) - 1|^2$).

(iv) Finally, the gradients of the loss function with respect to the trainable weights of the model are then computed using the top level *tf.GradientTape*, and the *Adam* optimizer is used to update the weights and biases based on these gradients. Finally, the model metrics (loss and MSE) are updated, and the method returns a dictionary of the updated metrics.

```
class ODE_1st(tf.keras.Model):
    def train_step(self, data):
        # Training points
        #and the analytical (exact) solution at these points
        x, y_exact = data
        # Initial conditions for the PINN
        x0=tf.constant([0.0], dtype=tf.float32)
        y0_exact=tf.constant([1.0], dtype=tf.float32)
        # Calculate the gradients and update weights and bias
        with tf.GradientTape() as tape:
            # Calculate the gradients dy/dx
            with tf.GradientTape() as tape2:
              tape2.watch(x0)
              tape2.watch(x)
              y0_NN = self(x0, training=True)
              y_NN  = self(x, training=True)
            dy_dx_NN= tape2.gradient(y_NN,x)
            #Loss= ODE+ boundary/initial conditions
            loss=self.compiled_loss(dy_dx_NN, -y_NN)\
                +self.compiled_loss(y0_NN,y0_exact)
        gradients = tape.gradient(loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(gradients, self.
                                           trainable_weights))
        self.compiled_metrics.update_state(y_exact, y_NN)
        return {m.name: m.result() for m in self.metrics}
```

2.3. Run the PINN

The code of this section defines and trains the NN model to solve the differential equation using the PINN approach. Here's what the code does:

(i) *n_train*, *xmin*, and *xmax* define the number of training points and the range of the input values.

(ii) *x_train* is a 1D NumPy array of size *n_train* containing equally spaced input values between *xmin* and *xmax*. $x0$ is the initial condition for the PINN, which is set to 0.0. The first element of *x_train* is replaced by $x0$.

(iii) *y_train* is a 1D tensor of size *n_train* containing the true solution to the differential equation at the corresponding input values in *x_train*. The true solution is computed using the *tf.exp* function in this example.

(iv) The NN model is defined with an input layer, three hidden layers with 50 neurons each, and an output layer with a single output neuron. Note that the activation function of all neurons is *elu*, except for the last one, which does not apply any activation function (*activation=None*).

(v) The model is compiled using: i) MSE as the loss function (here is "written" the ODE and its initial conditions through the custom Keras model class *ODE_1st*, as defined in the previous section); ii) the Adam optimizer with a learning rate of 0.001; and iii) the MSE metrics (distance between y_{NN} and y_{exact}).

(vi) The *model.fit* method is used to train the model for 50 epochs, with a batch size of 1, using *x_train* as inputs. Note that *y_train* is used only to estimate the accuracy of the predicted solution, and it has not been used in the training process (PINNs make not use of external data to work). Therefore, this requires to know the analytical solution beforehand.

(vii) Finally, the *history* object returned by *model.fit* contains information about the training progress, such as the total loss and metric values at each epoch.

```
n_train = 20
xmin = 0
xmax = 4

# Definition of the function domain
x_train=np.linspace(xmin,xmax,n_train)

# The real solution y(x) for training evaluation
y_train=tf.exp(-x_train)

# Input and output neurons (from the data)
input_neurons  = 1
output_neurons = 1

# Hiperparameters
epochs = 40

# Definition of the the model
activation='elu'
input=Input(shape=(input_neurons,))
x=Dense(50, activation=activation)(input)
x=Dense(50, activation=activation)(x)
x=Dense(50, activation=activation)(x)
output = Dense(output_neurons,activation=None)(x)
model=ODE_1st(input,output)

# Definition of the metrics, optimizer and loss
loss= tf.keras.losses.MeanSquaredError()
metrics=tf.keras.metrics.MeanSquaredError()
optimizer= Adam(learning_rate=0.001)

model.compile(loss=loss,
          optimizer=optimizer,
          metrics=[metrics])
model.summary()

history=model.fit(x_train, y_train,batch_size=1,epochs=epochs)
```

The table 1 summarized the hyperparameters used with this PINN.

Parameter	Value
interval	(0,4)
n train	20
n hidden layers	3
n neurons/layer	1,50,50,50,1
epochs	50
activation	elu
optimizer	adam
learning rate	0,001

Table 1: Hyperparameters of the PINN to solve equation (1).

Figure 1 provides an overview of the neural network architecture, which consists of 5251 trainable parameters. The shape of the outputs from each layer and their corresponding trainable parameters are also presented in the image. The image was generated with the *model.summary*() method of Keras.

```
Model: "ode_1st"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1)]               0

 dense (Dense)               (None, 50)                100

 dense_1 (Dense)             (None, 50)                2550

 dense_2 (Dense)             (None, 50)                2550

 dense_3 (Dense)             (None, 1)                 51

=================================================================
Total params: 5,251
Trainable params: 5,251
Non-trainable params: 0
```

Figure 1: Keras-Tensorflow model of the PINN as given by the *model.summary*() method of Keras.

2.4. Evolution of losses during training

Using the code below, the evolution of metrics as a function of the number of epochs can be obtained graphically.

```
# summarize history for loss and metris
plt.rcParams['figure.dpi'] = 150
plt.plot(history.history['loss'],color='magenta',
         label='Total losses ($L_D + L_B$)')
plt.plot(history.history['mean_squared_error'],color='cyan',label='MSE')
plt.yscale("log")
plt.xlabel('epochs')
plt.legend(loc='upper right')
plt.show()
```

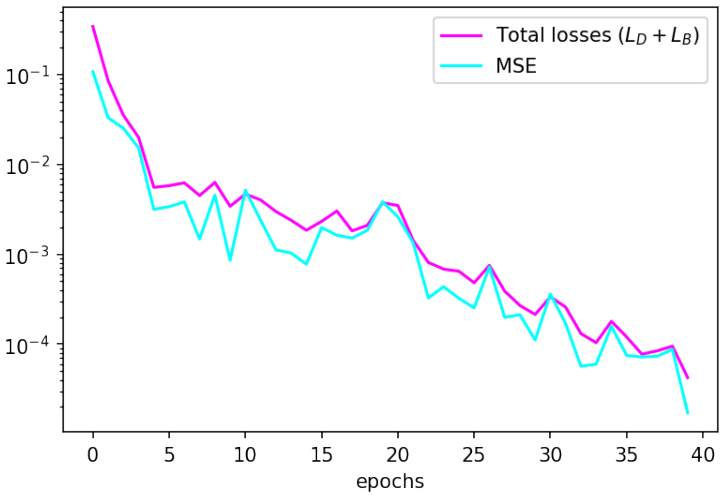

Figure 2: Training evolution as a function of the number of epochs. Two indicators of the performance of the NN are plotted: the total loss ($L_D + L_B$) and the MSE. The last measures the distance per point between $y_{analytic}$ and y_{NN}, evaluated at the training points.

2.5. Solution and its derivatives

TensorFlow allows the AD of any function $y(x)$, making it possible to calculate all of its derivatives. Since $y_{NN}(x)$ is simply a function, we can straightforwardly obtain them using this feature.

In the code, we define a set of validation points. Points not previously seen by the NN during training. The exact (analytical) values of $y(x)$ are also obtained.

The two first derivatives are obtained with the help of two tf.GradientTape environments. These lines of the code look like the ones used in the Definition of the PINN (Section 2.2).

```
# Check the PINN at different points not included in the training set
n = 500
x=np.linspace(0,4,n)
y_exact=tf.exp(-x)
```

```
y_NN=model.predict(x)

# The gradients (y'(x) and y''(x)) from the model
x_tf = tf.convert_to_tensor(x, dtype=tf.float32)
with tf.GradientTape(persistent=True) as t:
  t.watch(x_tf)
  with tf.GradientTape(persistent=True) as t2:
        t2.watch(x_tf)
        y = model(x_tf)
  dy_dx_NN = t2.gradient(y, x_tf)
d2y_dx2_NN = t.gradient(dy_dx_NN, x_tf)

# Plot the results
plt.rcParams['figure.dpi'] = 150
plt.plot(x, y_exact, color="black",linestyle='solid',
                     linewidth=2.5,label="$y(x)$ analytical")
plt.plot(x, y_NN, color="red",linestyle='dashed',
                     linewidth=2.5, label="$y_{NN}(x)$")
plt.plot(x, dy_dx_NN, color="blue",linestyle='-.',
                     linewidth=3.0, label="$y'_{NN}(x)$")
plt.plot(x, d2y_dx2_NN, color="green", linestyle='dotted',
                     linewidth=3.0, label="$y''_{NN}(x)$")
plt.legend()
plt.xlabel("x")
plt.show()
```

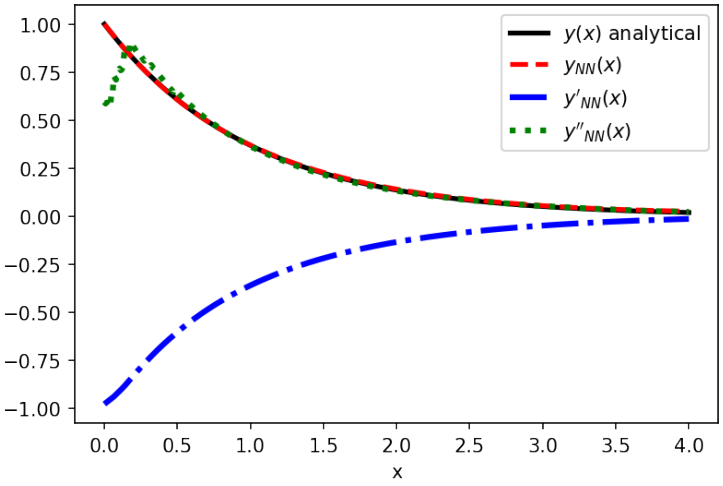

Figure 3: The analytical solution and the prediction for $y(x)$ and its derivatives as a function of x. The results are obtained from previously unseen points inside and outside the training interval, which were utilized as validation data for the PINN.

3. Example 2: 2nd order linear Ordinary Differential Equations

Here we solve the second example described in the paper, the solution to the harmonic oscillator problem using PINNs:

$$\begin{cases} y''(x) + y(x) = 0 \quad with \quad 0 \leqslant x \leqslant 8 \\ y(0) = 1, y'(0) = 0 \end{cases} \tag{2}$$

, whose analytic solution is $y(x) = \cos(x)$.

The codes for the 2nd-order ODE examples are analog to the example of the previous section. However, we need to pay special attention to the definition of the total loss.

3.1. Main libraries

We use here the same list of libraries that in the first example.

```
# Tensorflow Keras and the rest of the packages
import tensorflow as tf
from tensorflow.keras.layers import Dense,Input
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt
```

3.2. Definition of the PINN

As in the previous case, this is the most technical step in the code where the loss is defined, so it incorporates the differential equation.

We have introduced several significant changes to our approach. Firstly, we utilized an extra *tf.GradientTape* to calculate the second-order derivative required to define the loss. Additionally, we incorporated a new term to account for the initial velocity condition, and we made modifications to the differential equation term. Specifically, we are now solving for $y'' + y = 0$, and as a result, we are utilizing $self.compiled\_loss(d2y\_dx2, -y)$ in our computations.

```
class ODE_2nd(tf.keras.Model):
    def train_step(self, data):
        # Training points and the analytical
        # (exact) solution at this points
        x, y_exact = data
        #Change initial conditions for the PINN
        x0=tf.constant([0.0], dtype=tf.float32)
        y0_exact=tf.constant([1.0], dtype=tf.float32)
        dy_dx0_exact=tf.constant([0.0], dtype=tf.float32)
        # Calculate the gradients and update weights and bias
        with tf.GradientTape() as tape:
            tape.watch(x)
            tape.watch(y_exact)
            tape.watch(x0)
            tape.watch(y0_exact)
            tape.watch(dy_dx0_exact)
            # Calculate the gradients dy2/dx2, dy/dx
            with tf.GradientTape() as tape0:
                    tape0.watch(x0)
                    y0_NN = self(x0,training=False)
                    tape0.watch(y0_NN)
            dy_dx0_NN = tape0.gradient(y0_NN, x0)
            with tf.GradientTape() as tape1:
                tape1.watch(x)
                with tf.GradientTape() as tape2:
                    tape2.watch(x)
                    y_NN = self(x,training=False)
                    tape2.watch(y_NN)
                dy_dx_NN = tape2.gradient(y_NN, x)
                tape1.watch(y_NN)
```

```
            tape1.watch(dy_dx_NN)
        d2y_dx2_NN = tape1.gradient(dy_dx_NN, x)
        tape.watch(y_NN)
        tape.watch(dy_dx_NN)
        tape.watch(d2y_dx2_NN)

        #Loss= ODE+ boundary/initial conditions
        y0_exact=tf.reshape(y0_exact,shape=y_NN[0].shape)
        dy_dx0_exact=tf.reshape(dy_dx0_exact,shape=dy_dx0_NN.shape)

        loss= self.compiled_loss(y0_NN,y0_exact)\
              +self.compiled_loss(d2y_dx2_NN,-y_NN)\
              +self.compiled_loss(dy_dx0_NN,dy_dx0_exact)

    gradients = tape.gradient(loss, self.trainable_weights)
    self.optimizer.apply_gradients(zip(gradients, self.
                                       trainable_weights))
    self.compiled_metrics.update_state(y_exact, y_NN)
    return {m.name: m.result() for m in self.metrics}
```

3.3. Run the PINN

We used in solving the 2nd-order ODE an specific *callback* of Keras, which stops the training if there is no improvement and saves the NN's configuration with the lowest loss.

Here, we also introduced a different initializer of the weights and biases called GlorotUniform. There are many initializers, and part of improving the NN predictions consists of finding the best initializer in each case.

```
n_train = 18
xmin = 0.0
xmax = 8.0

# Definition of the function domain
x_train=np.linspace(xmin,xmax,n_train)

# The solution y(x) for training and validation datasets
y_train=np.cos(x_train)

# Input and output neurons (from the data)
input_neurons  = 1
output_neurons = 1

# Hiperparameters
epochs = 500

# Definition of the the model
initializer = tf.keras.initializers.GlorotUniform(seed=5)
activation='tanh'
input=Input(shape=(input_neurons,))
x=Dense(50, activation=activation,
            kernel_initializer=initializer)(input)
x=Dense(50, activation=activation,
            kernel_initializer=initializer)(x)
x=Dense(50, activation=activation,
            kernel_initializer=initializer)(x)
output = Dense(output_neurons,
               activation=None,
               kernel_initializer=initializer)(x)
```

```
model=ODE_2nd(input,output)

# Definition of the metrics, optimizer and loss
loss= tf.keras.losses.MeanSquaredError()
metrics=tf.keras.metrics.MeanSquaredError()
optimizer= Adam(learning_rate=0.001)

model.compile(loss=loss,
          optimizer=optimizer,
          metrics=[metrics])
model.summary()

# Just use 'fit' as usual
callback = tf.keras.callbacks.EarlyStopping(monitor='loss',
                                            patience=1000,
                                            restore_best_weights=True)

history=model.fit(x_train, y_train,batch_size=1, epochs=epochs,
                  callbacks=callback)
```

The hyperparameters for the 2nd-order ODE are summarized in Table 2, with slight modifications compared to the previous example.

Parameter	Value
interval	(0,8)
n train	18
n hidden layers	3
n neurons/layer	1,50,50,50,1
epochs	500
activation	tanh
optimizer	adam
learning rate	0,001

Table 2: Hyperparameters of the PINN to solve equation (2).

3.4. Evolution of losses during training

A comparable evolution, similar to the one depicted in the article in Fig. 3, can be observed by utilizing the same code as presented in Section 2.4.

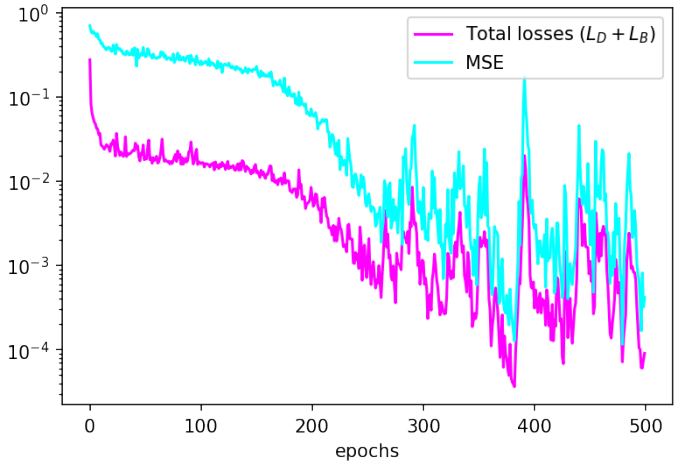

Figure 4: Evolution of losses during training, as a function of the number of epochs.

3.5. Solution and its derivatives

Using the code of Section 2.5, results like those shown in the paper in Fig. 3 can be obtained. It is important to note that the exact solution for the 2n-order ODE must be employed in this case, which is:

```
y_exact=tf.cos(-x)
```

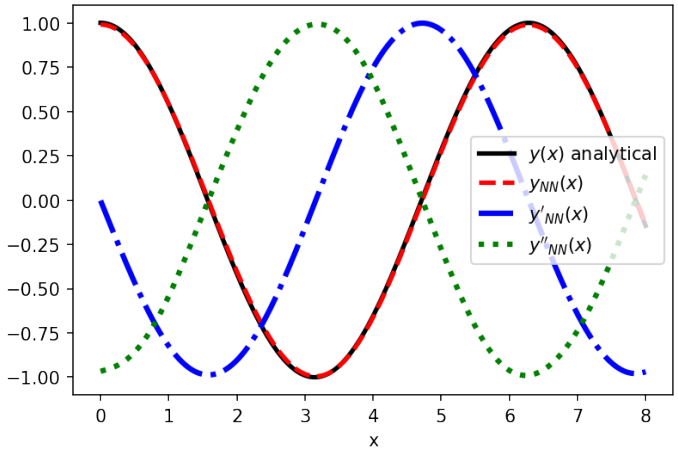

Figure 5: The analytical solution and the prediction for $y(x)$ and its derivatives as a function of x.

4. Example 3: 2nd order non-linear Ordinary Differential Equations

Finally, we describe the Python code to solve the 2nd-order nonlinear ODE used as an example in the article.

$$\begin{cases} y''(x) - y(x) - 3y^2(x) = 0 \\ y(0) = -1/2, y'(0) = 0 \end{cases} \tag{3}$$

, being its analytical solution $y(x) = -\frac{1}{2} \cdot sech^2(\frac{x}{2})$.

4.1. Main libraries

These are the same packages as those used in the previous examples.

```
# Tensorflow Keras and rest of the packages
import tensorflow as tf
from tensorflow.keras.layers import Dense,Input
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt
```

4.2. Definition of the PINN

As in the previous case, this is the most technical step in the code where the loss is defined, so it incorporates the differential equation.

```
class ODE_2nd(tf.keras.Model):
    def train_step(self, data):
        # Training points and the analytical
        # (exact) solution at this points
        x, y_exact = data
        #Change initial conditions for the PINN
        x0=tf.constant([0.0], dtype=tf.float32)
        y0_exact=tf.constant([-0.5], dtype=tf.float32)
        dy_dx0_exact=tf.constant([0.0], dtype=tf.float32)
        C=tf.constant([1.0], dtype=tf.float32)
        # Calculate the gradients and update weights and bias
        with tf.GradientTape() as tape:
            tape.watch(x)
            tape.watch(y_exact)
            tape.watch(x0)
            tape.watch(y0_exact)
            tape.watch(dy_dx0_exact)
            # Calculate the gradients dy2/dx2, dy/dx
            with tf.GradientTape() as tape0:
                    tape0.watch(x0)
                    y0_NN = self(x0,training=False)
                    tape0.watch(y0_NN)
            dy_dx0_NN = tape0.gradient(y0_NN, x0)
            with tf.GradientTape() as tape1:
                tape1.watch(x)
                with tf.GradientTape() as tape2:
                    tape2.watch(x)
                    y_NN = self(x,training=False)
                    tape2.watch(y_NN)
                dy_dx_NN = tape2.gradient(y_NN, x)
                tape1.watch(y_NN)
                tape1.watch(dy_dx_NN)
            d2y_dx2_NN = tape1.gradient(dy_dx_NN, x)
```

```
            tape.watch(y_NN)
            tape.watch(dy_dx_NN)
            tape.watch(d2y_dx2_NN)

            #Loss= ODE+ boundary/initial conditions
            y0_exact=tf.reshape(y0_exact,shape=y_NN[0].shape)
            dy_dx0_exact=tf.reshape(dy_dx0_exact,shape=dy_dx0_NN.shape)
            C=tf.reshape(C,shape=d2y_dx2_NN.shape)

            loss= self.compiled_loss(y0_NN,y0_exact)\
                  +self.compiled_loss(dy_dx0_NN,dy_dx0_exact)\
                  +self.compiled_loss(d2y_dx2_NN,C*y_NN+3.0*y_NN**2)

        gradients = tape.gradient(loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(gradients, self.
                                           trainable_weights))
        self.compiled_metrics.update_state(y_exact, y_NN)
        return {m.name: m.result() for m in self.metrics}
```

4.3. Run the PINN

Regarding the solution found for the 2nd-order ODE, the number of epochs increases because the learning rate is reduced. This example, however, is not so sensitive to the initial values of weights and biases, so we removed the GlorotUniform initializer.

```
n_train = 11
xmin = -5
xmax = 5

# Definition of the function domain
x_train=np.linspace(xmin,xmax,n_train)

# The solution y(x) for training and validation datasets
x0=0.0
y_train=-0.5*1.0*(1.0/np.cosh(0.5*np.sqrt(1.0)*(x_train-x0)))**2

# Input and output neurons (from the data)
input_neurons  = 1
output_neurons = 1

# Hiperparameters
epochs = 2000

# Definition of the model
activation='tanh'
input=Input(shape=(input_neurons,))
x=Dense(50, activation=activation)(input)
x=Dense(50, activation=activation)(x)
x=Dense(50, activation=activation)(x)
output = Dense(output_neurons,activation=None)(x)
model=ODE_2nd(input,output)

# Definition of the metrics, optimizer and loss
loss=tf.keras.losses.MeanSquaredError()
metrics=tf.keras.metrics.MeanSquaredError()
optimizer= Adam(learning_rate=0.0001)

model.compile(loss=loss,
          optimizer=optimizer,
          metrics=[metrics])
```

```
model.summary()

# Just use 'fit' as usual
callback = tf.keras.callbacks.EarlyStopping(monitor='loss',
                                            patience=1000,
                                            restore_best_weights=True)

history=model.fit(x_train, y_train, batch_size=1, epochs=epochs,
                  callbacks=callback)
```

Table 3 sums up the hyperparameters for this ODE.

Parameter	Value
interval	(-5,5)
n train	11
n hidden layers	3
n neurons/layer	1,50,50,50,1
epochs	2000
activation	tanh
optimizer	adam
learning rate	0,0001

Table 3: Hyperparameters of the PINN to solve equation (3).

4.4. Evolution of losses during training

A comparable evolution, similar to the one depicted in the article in Fig. 3, can be observed by utilizing the same code as presented in Section 2.4.

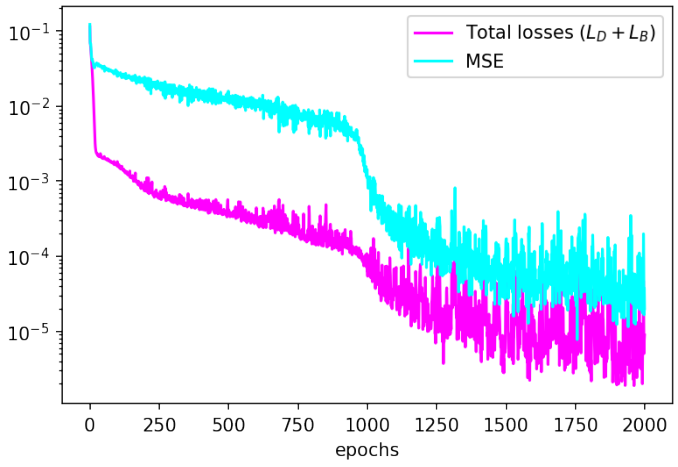

Figure 6: Training evolution as a function of the number of epochs.

4.5. Solution and its derivatives

To obtain comparable results to those presented in Fig. 3 of the article, one can use the code from Section2.5. However, it is necessary to include the exact solution of the 2nd-order nonlinear ODE:

```
x0=0.0
y=-0.5*1.0*(1.0/np.cosh(0.5*np.sqrt(1.0)*(x-x0)))**2
```

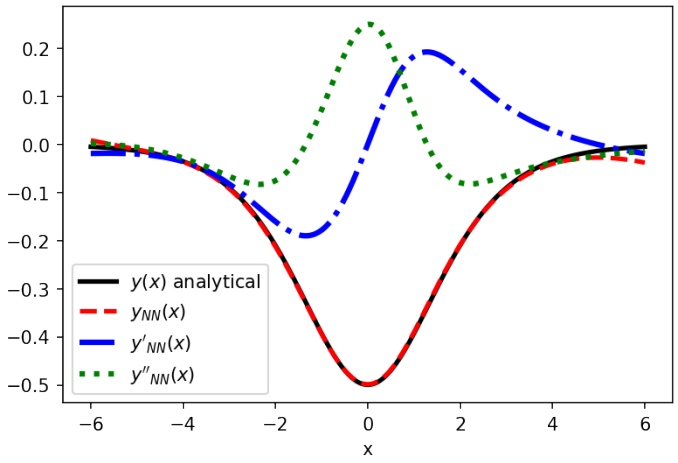

Figure 7: The analytical solution and the prediction for $y(x)$ and its derivatives as a function of x.

5. Performance of the PINNs

We conducted 100 simulations for each example, enforcing a total loss threshold of $1.0 \cdot 10^{-4}$ for Examples 1 and 2 and $1.0 \cdot 10^{-5}$ for Example 3.

The numerical experiments are presented in Figure 8, which indicate that, on average, the threshold is reached after 42, 326, and 1422 epochs. However, it's worth noting that the distributions are non-normal, and a more insightful analysis requires the use of median and mode statistics, where the median represents the middle value of the dataset. The mode defines the most common value. For instance, the median and mode for the Example 1 are 40 and 39 epochs. For the Example 2 we have 317 and 312. Finally, for the Example 3, the median and mode are 1256 and 1225 respectively.

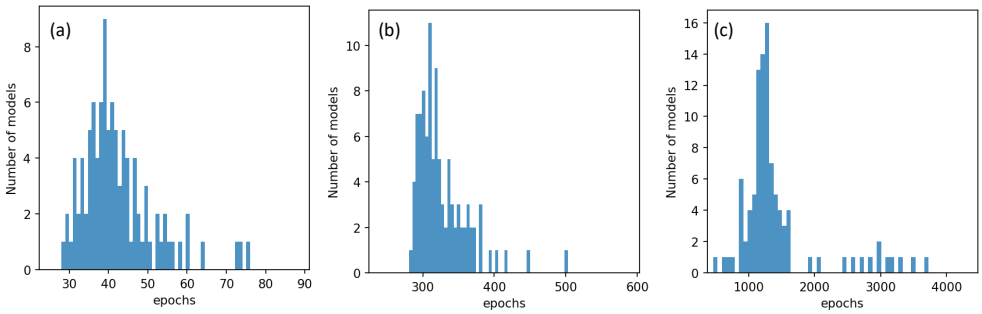

Figure 8: The histograms display the distribution of the number of trained models that achieved the convergence threshold within a specific number of epochs. This data was obtained for: (a) the 1st-order ODE studied, corresponding to the exponential decay solution; (b) the 2nd-order ODE (harmonic oscillator); and (c) the 2nd-order nonlinear ODE (Korteweg-de Vries equation).

6. References

[1] Luis Medrano Navarro, Luis Martín Moreno, Sergio G Rodrigo'. PINNs-for-education. `https://github.com/IrisFDTD/PINNs-for-education`, 2023.

[2] Francois Chollet. Deep learning with python. Manning Publications. *New York*, 2017.

[3] Aurèlien Gèron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc., Sebastopol, 2019.