neurodiffeq Documentation

NeuroDiffGym

Dec 07, 2022

Contents:

CHAPTER 1

Introduction

Differential equations emerge in various scientific and engineering domains for modeling physical phenomena. Most differential equations of practical interest are analytically intractable. Traditionally, differential equations are solved by numerical methods. Sophisticated algorithms exist to integrate differential equations in time and space. Time integration techniques continue to be an active area of research and include backward difference formulas and Runge-Kutta methods. Common spatial discretization approaches include the finite difference method (FDM), finite volume method (FVM), and finite element method (FEM) as well as spectral methods such as the Fourier-spectral method. These classical methods have been studied in detail and much is known about their convergence properties. Moreover, highly optimized codes exist for solving differential equations of practical interest with these techniques. While these methods are efficient and well-studied, their expressibility is limited by their function representation.

Artificial neural networks (ANN) are a framework of machine learning algorithms that use a collection of connected units to learn function mappings. The most basic form of ANNs, multilayer perceptrons, have been proven to be universal function approximators. This suggests the possibility of using ANNs to solve differential equations. Previous studies have demonstrated that ANNs have the potential to solve ordinary differential equations (ODEs) and partial differential equations (PDEs) with certain initial/boundary conditions. These methods show nice properties including (1) continuous and differentiable solutions, (2) good interpolation properties, (3) less memory-intensive. By less memory-intensive we mean that only the weights of the neural network have to be stored. The solution can then be recovered anywhere in the solution domain because a trained neural network is a closed form solution.

Given the interest in developing neural networks for solving differential equations, it would be extremely beneficial to have an easy-to-use software package that allows researchers to quickly set up and solve problems.

1.1 Differential Equations

Differential equations can be divided into 2 types: ordinary differential equations (ODEs) and partial differential equations (PDEs).

1.1.1 Ordinary Differential Equations

An ordinary differential equation (ODE) is an differential equation that contains only one independent variable (a scalar). Let $t \in \mathbb{R}$ be the independent variable and $x : \mathbb{R} \to \mathbb{R}$ be a function of t. An ordinary differential equation of

order n takes the form:

$$F(t, x, \frac{dx}{dt}, \frac{d^2x}{dt^2}, \ldots, \frac{d^nx}{dt^n}) = 0,$$

A general solution of an nth-order equation is a solution containing n arbitrary independent constants of integration. A particular solution is derived from the general solution by setting the constants to particular values. This is often done by imposing an initial condition or boundary condition to the ODE. The former corresponds to an initial value problem (IVP) and the latter a boundary value problem (BVP).

Initial Value Problems

For the following ODE:

$$F(t, x, \frac{dx}{dt}, \frac{d^2x}{dt^2}, \ldots, \frac{d^nx}{dt^n}) = 0,$$

If we specify that

$$x(t_0) = x_0,$$

then we have an initial value problem. Initial value problem can be seen as the question of how x will evolve with time given $x = x_0$ at time $t = t_0$.

Boundary Value Problems

A boundary value problem has conditions specified at the boundaries of the independent variables. In the context of ordinary differential equations, a boundary problem is one that put some restrictions on x at the initial t and final t. There are several kinds of boundary conditions.

For the following ODE:

$$F(t, x, \frac{dx}{dt}, \frac{d^2x}{dt^2}, \ldots, \frac{d^nx}{dt^n}) = 0,$$

If we specify that

$$x(t_{ini}) = f,$$
$$x(t_{fin}) = g,$$

then we have a Dirichlet boundary condition.

If we specify that

$$\left.\frac{dx}{dt}\right|_{t=t_{ini}} = f,$$
$$\left.\frac{dx}{dt}\right|_{t=t_{fin}} = g,$$

then we have a Neumann boundary condition.

If we specify that

$$x(t_{ini}) + \left.\frac{dx}{dt}\right|_{t=t_{ini}} = f,$$
$$x(t_{fin}) + \left.\frac{dx}{dt}\right|_{t=t_{fin}} = g,$$

then we have a Robin boundary condition.

Boundary conditions of mixed types can also be specified on a different subset of the boundaries (In this case, that will be one boundary condition for $t = t_{ini}$ and another boundary condition of a different type for $t = t_{fin}$).

1.1.2 System of Ordinary Differential Equations

A number of coupled differential equations form a system of equations. Let $t \in \mathbb{R}$ be the independent variable and $\vec{x} : \mathbb{R} \to \mathbb{R}^{>}$ be a function of t. A system of ordinary differential equations of order n takes the form:

$$F(t, \vec{x}, \frac{d\vec{x}}{dt}, \frac{d^2\vec{x}}{dt^2}, \ldots, \frac{d^n\vec{x}}{dt^n}) = \vec{0},$$

This can be written in matrix form as

$$\begin{pmatrix} f_0(t, \vec{x}, \frac{d\vec{x}}{dt}, \frac{d^2\vec{x}}{dt^2}, \ldots, \frac{d^n\vec{x}}{dt^n}) \\ f_1(t, \vec{x}, \frac{d\vec{x}}{dt}, \frac{d^2\vec{x}}{dt^2}, \ldots, \frac{d^n\vec{x}}{dt^n}) \\ \vdots \\ f_{m-1}(t, \vec{x}, \frac{d\vec{x}}{dt}, \frac{d^2\vec{x}}{dt^2}, \ldots, \frac{d^n\vec{x}}{dt^n}) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

1.2 Partial Differential Equations

A partial differential equation (PDE) is a differential equation that contains multiple independent variables. ODEs can be seen as a special case of PDEs. A partial differential equation of $u(x_1 \ldots x_n)$ with n independent variables takes the form:

$$F(x_1, \ldots x_n; u, \frac{\partial u}{\partial x_1}, \ldots \frac{\partial u}{\partial x_n}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots \frac{\partial^2 u}{\partial x_1 \partial x_n}; \ldots) = 0.$$

Similar as PDE, a general solution of a PDE contains arbitrary independent constants of integration. A particular solution is derived from the general solution by setting the constants to particular values. To fix to a particular solution, we need to impose initial value conditions and boundary value conoditions as well.

1.3 About NeuroDiffEq

`NeuroDiffEq` is a Python package built with `PyTorch` that uses ANNs to solve ordinary and partial differential equations (ODEs and PDEs). It is designed to encourage the user to focus more on the problem domain (What is the differential equation we need to solve? What are the initial/boundary conditions?) and at the same time allow them to dig into solution domain (What ANN architecture and loss function should be used? What are the training hyperparameters?) when they want to. `NeuroDiffEq` can solve a variety of canonical PDEs including the heat equation and Poisson equation in a Cartesian domain with up to two spatial dimensions. We are actively working on extending `NeuroDiffEq` to support three spatial dimensions. `NeuroDiffEq` can also solve arbitrary systems of nonlinear ordinary differential equations.

1.4 Two examples

1.4.1 Lotka–Volterra equations

The Lotka–Volterra equations are a system of first-order, nonlinear ODEs that have been used to model predator-prey dynamics in biological systems as well as problems in chemical kinetics. They are given by:

$$\begin{aligned} \frac{dx(t)}{dt} &= \alpha x(t) - \beta x(t) y(t), \quad x(0) = x_0 \\ \frac{dy(t)}{dt} &= \delta x(t) y(t) - \gamma y(t), \quad y(0) = y_0. \end{aligned}$$

The time-evolution of the population of the prey and predator are given by x and y, respectively, with x_0 and y_0 the initial populations. The coupling parameters α, β, δ and γ describe the interaction of the two species. Let $\alpha = \beta = \delta = \gamma = 1$, $x_0 = 1.5$, and $y_0 = 1.0$. For comparison purposes, we solve this problem numerically with `scipy` and `NeuroDiffEq`. Figure 1 compares the predator and prey populations from the numerical integrator and the neural network. The solutions are qualitatively the same.

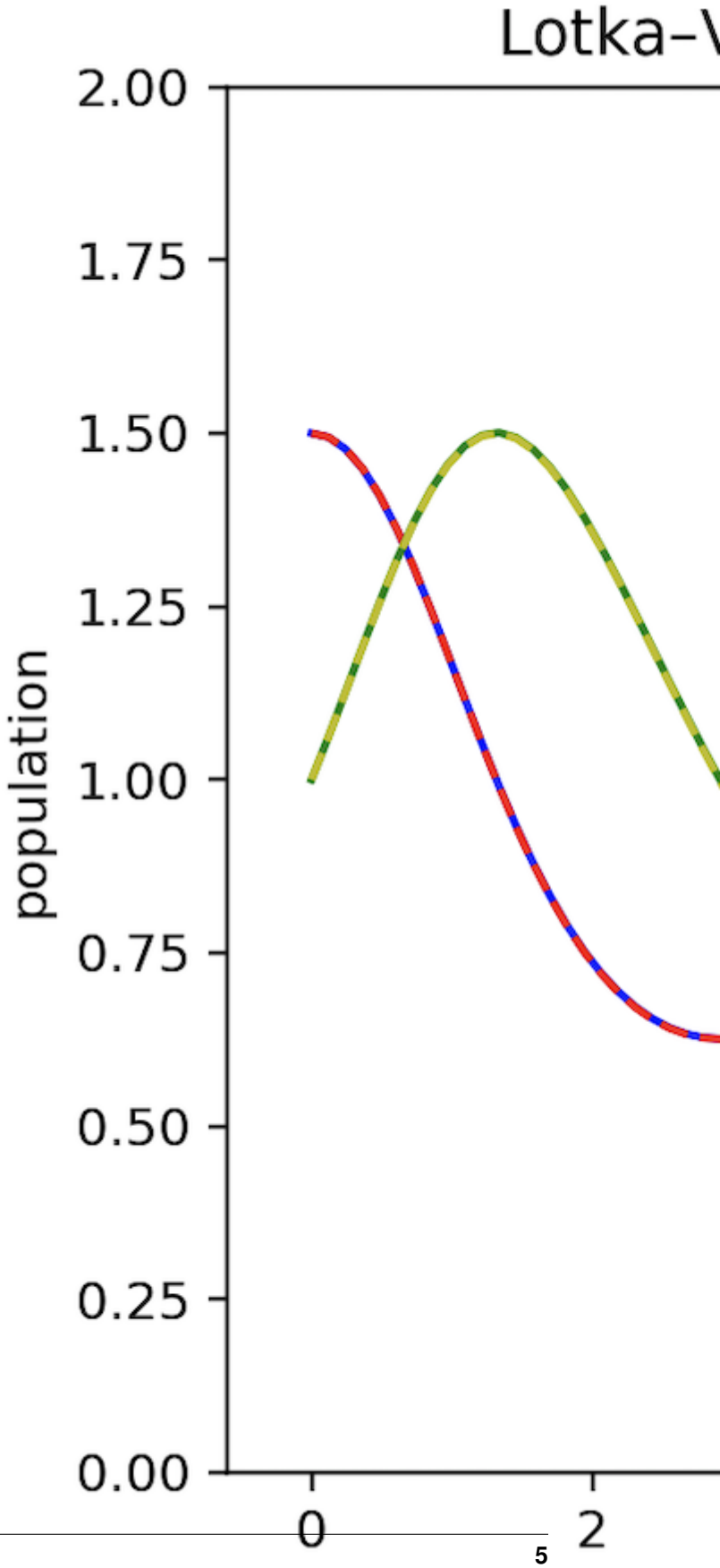

Lotka–V
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
population
0
2

Figure 1: Comparing numerical and ANN-based solutions of Lotka–Volterra equations.

1.4.2 Poisson's equation

Poisson's equation is a second-order linear PDE. It can be used to describe the potential field caused by a given charge or mass density distribution. In two dimensional Cartesian coordinates, it takes the form:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) u(x, y) = f(x, y).$$

We solve the equation in the domain $(x, y) \in (0, 1) \times (0, 1)$ with homogeneous Dirichlet boundary conditions,

$$u(x, 0) = u(x, 1) = u(0, y) = u(1, y) = 0.$$

With $f(x, y) = 2x(y - 1)(y - 2x + xy + 2)e^{x-y}$ the analytical solution is

$$u(x, y) = x(1 - x)y(1 - y)e^{x-y}.$$

Figure 2 presents contours of the neural network solution (left), the analytical solution (middle), and the error between the analytical and neural network solution (right). The largest error in the neural network solution is around $6 \cdot 10^{-5}$.

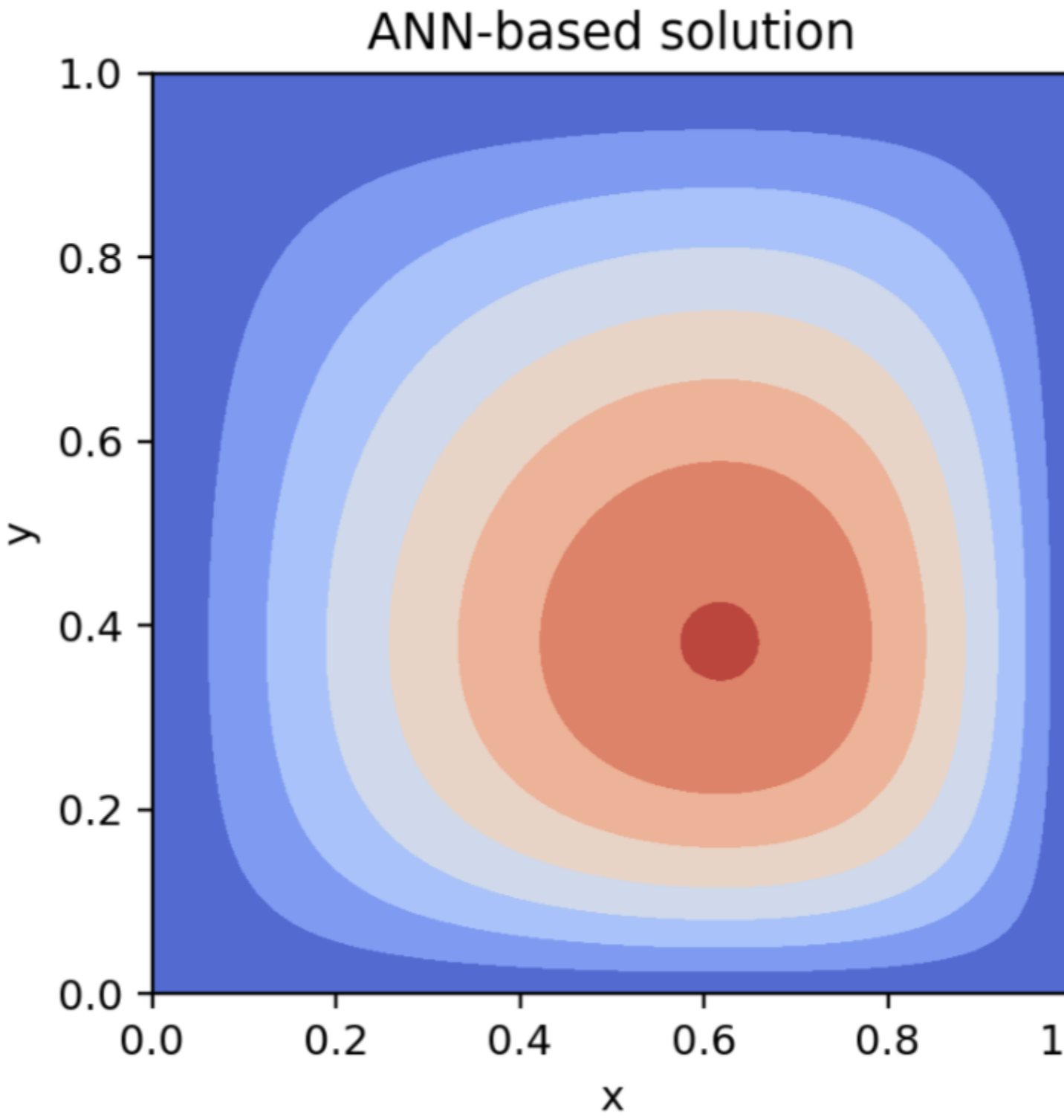


Figure 2: Comparing analytical and ANN-based solutions of Poisson's equation.

[ ]:

CHAPTER 2

Getting Started

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
```

2.1 Solving ODEs

There are two ways to solve an ODE (or ODE system). 1. As a legacy option, ODEs can be solved by `neurodiffeq.ode.solve`. 2. For users who want fine-grained control over the training process, please consider using a `neurodiffeq.solvers.Solver1d`.

- The first option is easier to use but has been **deprecated** and might be removed in a future version.
- The second option is **recommended** for most users and supports advanced features like custom callbacks, checkpointing, early stopping, gradient clipping, learning rate scheduling, curriculum learning, etc.

Just for the sake of notation in the following examples, here we see differentiation as an operation, then an ODE can be rewritten as

$$F(x, t) = 0.$$

2.1.1 ODE Example 1: Exponential Decay (using the legacy `solve` function)

To show how simple `neurodiffeq` can be, we'll first introduce the legacy option with the `solve` function form `neurodiffeq.ode`.

Start by solving

$$\frac{du}{dt} = -u.$$

for $u(t)$ with $u(0) = 1.0$. The analytical solution is

$$u = e^{-t}.$$

For `neurodiffeq.ode.solve` to solve this ODE, the following parameters needs to be specified:

- `ode`: a function representing the ODE to be solved. It should be a function that maps (u, t) to $F(u, t)$. Here we are solving

$$F(u,t) = \frac{du}{dt} + u = 0,$$

  then `ode` should be `lambda u, t: diff(u, t) + u`, where `diff(u, t)` is the first order derivative of `u` with respect to `t`.

- `condition`: a `neurodiffeq.conditions.BaseCondition` instance representing the initial condition / boundary condition of the ODE. Here we use `neurodiffeq.conditions.IVP(t_0=0.0, u_0=1.0)` to ensure $u(0) = 1.0$.

- `t_min` and `t_max`: the domain of t to solve the ODE on.

[2]:
```
from neurodiffeq import diff      # the differentiation operation
from neurodiffeq.ode import solve # the ANN-based solver
from neurodiffeq.conditions import IVP   # the initial condition
```

[3]:
```
exponential = lambda u, t: diff(u, t) + u # specify the ODE
init_val_ex = IVP(t_0=0.0, u_0=1.0)       # specify the initial conditon

# solve the ODE
solution_ex, loss_ex = solve(
    ode=exponential, condition=init_val_ex, t_min=0.0, t_max=2.0
)
```

```
/Users/liushuheng/Documents/GitHub/neurodiffeq/neurodiffeq/ode.py:260: FutureWarning:␣
↪The `solve_system` function is deprecated, use a `neurodiffeq.solvers.Solver1D`␣
↪instance instead
  warnings.warn(
```

(Oops, we have a warning. As we have explained, the `solve` function still works but is deprecated. Hence we have the warning message, which we'll ignore for now.)

`solve` returns a tuple, where the first entry is the solution (as a function) and the second entry is the history (of loss and other metrics) of training and validation. The solution is a function that maps t to u. It accepts `numpy.array` or `troch.Tensor` as its input. The default return type of the solution is `torch.tensor`. If we wanted to return `numpy.array`, we can specify `to_numpy=True`. The history is a dictionary, where the 'train_loss' entry is the training loss and the 'valid_loss' entry is the validation loss. Here we compare the ANN-based solution with the analytical solution:

[4]:
```
ts = np.linspace(0, 2.0, 50)
u_net = solution_ex(ts, to_numpy=True)
u_ana = np.exp(-ts)

plt.figure()
plt.plot(ts, u_net, label='ANN-based solution')
plt.plot(ts, u_ana, '.', label='analytical solution')
plt.ylabel('u')
plt.xlabel('t')
plt.title('comparing solutions')
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[5]:
```
plt.figure()
plt.plot(loss_ex['train_loss'], label='training loss')
plt.plot(loss_ex['valid_loss'], label='validation loss')
plt.yscale('log')
plt.title('loss during training')
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

We may want to see the check the solution and the loss function during solving the problem (training the network). To do this, we need to pass a `neurodiffeq.monitors.Monitor1D` object to `solve`. A `Monitor1D` has the following parameters:

- `t_min` and `t_max`: the region of t we want to monitor
- `check_every`: the frequency of visualization. If `check_every=100`, then the monitor will visualize the solution every 100 epochs.

`%matplotlib notebook` should be executed to allow `Monitor1D` to work. Here we solve the above ODE again.

[6]:
```
from neurodiffeq.monitors import Monitor1D
```

[7]:
```
# This must be executed for Jupyter Notebook environments
# If you are using Jupyter Lab, try `%matplotlib widget`
# Don't use `%matplotlib inline`!

%matplotlib notebook

solution_ex, _ = solve(
    ode=exponential, condition=init_val_ex, t_min=0.0, t_max=2.0,
    monitor=Monitor1D(t_min=0.0, t_max=2.0, check_every=100)
)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
/Users/liushuheng/Documents/GitHub/neurodiffeq/neurodiffeq/ode.py:260: FutureWarning:␣
↪The `solve_system` function is deprecated, use a `neurodiffeq.solvers.Solver1D`␣
↪instance instead
  warnings.warn(
/Users/liushuheng/Documents/GitHub/neurodiffeq/neurodiffeq/solvers.py:389:␣
↪UserWarning: Passing `monitor` is deprecated, use a MonitorCallback and pass a list␣
↪of callbacks instead
  warnings.warn("Passing `monitor` is deprecated, "
```

Here we have two warnings. But **don't worry**, the training process is not affected.

- The first one warns that we should use a `neurodiffeq.solvers.Solver1D` instance, which we have discussed before.
- The second warning is slightly different. It says we should use a callback instead of using a `monitor`. Remember we said using a `neurodiffeq.solvers.Solver1D` allows flexible callbacks? This warning is also caused by using the deprecated `solve` function.

2.1.2 ODE Example 2: Harmonic Oscilator

Here we solve a harmonic oscillator:

$$F(u,t) = \frac{d^2u}{dt^2} + u = 0$$

for

$$u(0) = 0.0, \frac{du}{dt}|_{t=0} = 1.0$$

The analytical solution is

$$u = \sin(t)$$

We can include higher order derivatives in our ODE with the `order` keyword of `diff`, which is defaulted to 1.

Initial condition on $\frac{du}{dt}$ can be specified with the `u_0_prime` keyword of `IVP`.

[8]:
```
harmonic_oscillator = lambda u, t: diff(u, t, order=2) + u
init_val_ho = IVP(t_0=0.0, u_0=0.0, u_0_prime=1.0)
```

Here we will use another keyword for `solve`:

- `max_epochs`: the number of epochs to run

[9]:
```
solution_ho, _ = solve(
    ode=harmonic_oscillator, condition=init_val_ho, t_min=0.0, t_max=2*np.pi,
    max_epochs=3000,
    monitor=Monitor1D(t_min=0.0, t_max=2*np.pi, check_every=100)
)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
/Users/liushuheng/Documents/GitHub/neurodiffeq/neurodiffeq/ode.py:260: FutureWarning:␣
↪The `solve_system` function is deprecated, use a `neurodiffeq.solvers.Solver1D`␣
↪instance instead
  warnings.warn(
/Users/liushuheng/Documents/GitHub/neurodiffeq/neurodiffeq/solvers.py:389:␣
↪UserWarning: Passing `monitor` is deprecated, use a MonitorCallback and pass a list␣
↪of callbacks instead
  warnings.warn("Passing `monitor` is deprecated, "
```

This is the third time we see these warnings. I promise we'll learn to get ride of them by the end of this chapter :)

[10]:
```
ts = np.linspace(0, 2*np.pi, 50)
u_net = solution_ho(ts, to_numpy=True)
u_ana = np.sin(ts)

plt.figure()
plt.plot(ts, u_net, label='ANN-based solution')
plt.plot(ts, u_ana, '.', label='analytical solution')
plt.ylabel('u')
plt.xlabel('t')
plt.title('comparing solutions')
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.2 Solving Systems of ODEs

Systems of ODEs can be solved by `neurodiffeq.ode.solve_system`.

Again, just for the sake of notation in the following examples, here we see differentiation as an operation, and see each element $u_i$of $\vec{u}$ as different dependent vairables, then a ODE system above can be rewritten as

$$\begin{pmatrix} F_0(u_0, u_1, \ldots, u_{m-1}, t) \\ F_1(u_0, u_1, \ldots, u_{m-1}, t) \\ \vdots \\ F_{m-1}(u_0, u_1, \ldots, u_{m-1}, t) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

2.2.1 Systems of ODE Example 1: Harmonic Oscilator

For the harmonic oscillator example above, if we let $u_1 = u$ and $u_2 = \frac{du}{dt}$. We can rewrite this ODE into a system of ODE:

$$u_1' - u_2 = 0, \tag{2.1}$$
$$u_2' + u_1 = 0, \tag{2.2}$$
$$u_1(0) = 0, \tag{2.3}$$
$$u_2(0) = 1 \tag{2.4}$$

Here the analytical solution is

$$u_1 = \sin(t), \tag{2.5}$$
$$u_2 = \cos(t) \tag{2.6}$$

The `solve_system` function is for solving ODE systems. The signature is almost the same as `solve` except that we specify an `ode_system` and a set of `conditions`.

- `ode_system`: a function representing the system of ODEs to be solved. If the our system of ODEs is $f_i(u_0, u_1, ..., u_{m-1}, t) = 0$ for $i = 0, 1, ..., n-1$ where $u_0, u_1, ..., u_{m-1}$ are dependent variables and t is the independent variable, then `ode_system` should map $(u_0, u_1, ..., u_{m-1}, t)$ to a n-element list where the i^{th} element is the value of $f_i(u_0, u_1, ..., u_{m-1}, t)$.
- `conditions`: the initial value/boundary conditions as a list of Condition instance. They should be in an order such that the first condition constraints the first variable in f_i's (see above) signature (u_0). The second condition constraints the second (u_1), and so on.

[11]:
```
from neurodiffeq.ode import solve_system
```

[12]:
```
# specify the ODE system
parametric_circle = lambda u1, u2, t : [diff(u1, t) - u2,
                                        diff(u2, t) + u1]
# specify the initial conditions
init_vals_pc = [
    IVP(t_0=0.0, u_0=0.0),
```

(continues on next page)

(continued from previous page)

```
    IVP(t_0=0.0, u_0=1.0)
]

# solve the ODE system
solution_pc, _ = solve_system(
    ode_system=parametric_circle, conditions=init_vals_pc, t_min=0.0, t_max=2*np.pi,
    max_epochs=5000,
    monitor=Monitor1D(t_min=0.0, t_max=2*np.pi, check_every=100)
)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
/Users/liushuheng/Documents/GitHub/neurodiffeq/neurodiffeq/ode.py:260: FutureWarning:␣
↪The `solve_system` function is deprecated, use a `neurodiffeq.solvers.Solver1D`␣
↪instance instead
  warnings.warn(
/Users/liushuheng/Documents/GitHub/neurodiffeq/neurodiffeq/solvers.py:389:␣
↪UserWarning: Passing `monitor` is deprecated, use a MonitorCallback and pass a list␣
↪of callbacks instead
  warnings.warn("Passing `monitor` is deprecated, "
```

`solve_system` returns a tuple, where the first entry is the solution as a function and the second entry is the loss history as a list. The solution is a function that maps t to $[u_0, u_1, ..., u_{m-1}]$. It accepts `numpy.array` or `torch.Tensor` as its input.

Here we compare the ANN-based solution with the analytical solution:

[13]:
```
ts = np.linspace(0, 2*np.pi, 100)
u1_net, u2_net = solution_pc(ts, to_numpy=True)
u1_ana, u2_ana = np.sin(ts), np.cos(ts)

plt.figure()
plt.plot(ts, u1_net, label='ANN-based solution of $u_1$')
plt.plot(ts, u1_ana, '.', label='Analytical solution of $u_1$')
plt.plot(ts, u2_net, label='ANN-based solution of $u_2$')
plt.plot(ts, u2_ana, '.', label='Analytical solution of $u_2$')
plt.ylabel('u')
plt.xlabel('t')
plt.title('comparing solutions')
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.2.2 Systems of ODE Example 2: Lotka–Volterra equations

The previous examples are rather simple because they are both linear ODE systems. We have numerous existing numerical methods that solve these linear ODEs very well. To show the capability neurodiffeq, let's see this example of *nonlinear* ODEs.

Lotka–Volterra equations are a pair of nonlinear ODE frequently used to describe the dynamics of biological systems

in which two species interact, one as a predator and the other as prey:

$$\frac{du}{dt} = \alpha u - \beta uv \tag{2.7}$$
$$\frac{dv}{dt} = \delta uv - \gamma v \tag{2.8}$$

Let $\alpha = \beta = \delta = \gamma = 1$. Here we solve this pair of ODE when $u(0) = 1.5$ and $v(0) = 1.0$.

If not specified otherwise, `solve` and `solve_system` will use a fully-connected network with 1 hidden layer with 32 hidden units (tanh activation) to approximate each dependent variables. In some situations, we may want to use our own neural network. For example, the default neural net is not good at solving a problem where the solution oscillates. However, if we know in advance that the solution oscillates, we can use sin as activation function, which resulted in much faster convergence.

`neurodiffeq.FCNN` is a fully connected neural network. It is initiated by the following parameters:

- `hidden_units`: number of units in each hidden layer. If you have 3 hidden layers with 32, 64, and 16 neurons respectively, `hidden_units` should be a tuple `(32, 64, 16)`.
- `actv`: a `torch.nn.Module` *class*. e.g. `nn.Tanh`, `nn.Sigmoid`.

Here we will use another keyword for `solve_system`:

- `nets`: a list of networks to be used to approximate each dependent variable

[14]:
```
from neurodiffeq.networks import FCNN    # fully-connect neural network
from neurodiffeq.networks import SinActv # sin activation
```

[15]:
```
# specify the ODE system and its parameters
alpha, beta, delta, gamma = 1, 1, 1, 1
lotka_volterra = lambda u, v, t : [ diff(u, t) - (alpha*u  - beta*u*v),
                                    diff(v, t) - (delta*u*v - gamma*v), ]
# specify the initial conditions
init_vals_lv = [
    IVP(t_0=0.0, u_0=1.5),  # 1.5 is the value of u at t_0 = 0.0
    IVP(t_0=0.0, u_0=1.0),  # 1.0 is the value of v at t_0 = 0.0
]

# specify the network to be used to approximate each dependent variable
# the input units and output units default to 1 for FCNN
nets_lv = [
    FCNN(n_input_units=1, n_output_units=1, hidden_units=(32, 32), actv=SinActv),
    FCNN(n_input_units=1, n_output_units=1, hidden_units=(32, 32), actv=SinActv)
]

# solve the ODE system
solution_lv, _ = solve_system(
    ode_system=lotka_volterra, conditions=init_vals_lv, t_min=0.0, t_max=12,
    nets=nets_lv, max_epochs=3000,
    monitor=Monitor1D(t_min=0.0, t_max=12, check_every=100)
)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
/Users/liushuheng/Documents/GitHub/neurodiffeq/neurodiffeq/ode.py:260: FutureWarning:␣
↪The `solve_system` function is deprecated, use a `neurodiffeq.solvers.Solver1D`␣
↪instance instead
  warnings.warn(
```

(continues on next page)

(continued from previous page)

```
/Users/liushuheng/Documents/GitHub/neurodiffeq/neurodiffeq/solvers.py:389:␣
↪UserWarning: Passing `monitor` is deprecated, use a MonitorCallback and pass a list␣
↪of callbacks instead
  warnings.warn("Passing `monitor` is deprecated, "
```

2.2.3 Tired of the annoying warning messages? Let's get rid of them by using a 'Solver'

Now that you are familiar with the usage of `solve`, let's try the second way of solving ODEs – using a `neurodiffeq.solvers.Solver1D` instance. If you are familiar with `sklearn` or `keras`, the workflow with a *Solver* is quite similar.

1. Instantiate a solver. (Specify the ODE/PDE system, initial/boundary conditions, problem domain, etc.)
2. Fit the solver (Specify number of epochs to train, callbacks in each epoch, monitor, etc.)
3. Get the solutions and other internal variables.

This is the recommended way of solving ODEs (and PDEs later). Once you learn to use a Solver, you should stick to this way instead of using a `solve` function.

[16]:
```
from neurodiffeq.solvers import Solver1D

# Let's create a monitor first
monitor = Monitor1D(t_min=0.0, t_max=12.0, check_every=100)
# ... and turn it into a Callback instance
monitor_callback = monitor.to_callback()

# Instantiate a solver instance
solver = Solver1D(
    ode_system=lotka_volterra,
    conditions=init_vals_lv,
    t_min=0.1,
    t_max=12.0,
    nets=nets_lv,
)

# Fit the solver (i.e., train the neural networks)
solver.fit(max_epochs=3000, callbacks=[monitor_callback])

# Get the solution
solution_lv = solver.get_solution()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[17]:
```
ts = np.linspace(0, 12, 100)

# ANN-based solution
prey_net, pred_net = solution_lv(ts, to_numpy=True)

# numerical solution
from scipy.integrate import odeint
def dPdt(P, t):
    return [P[0]*alpha - beta*P[0]*P[1], delta*P[0]*P[1] - gamma*P[1]]
```

(continues on next page)

(continued from previous page)

```
P0 = [1.5, 1.0]
Ps = odeint(dPdt, P0, ts)
prey_num = Ps[:,0]
pred_num = Ps[:,1]

fig = plt.figure(figsize=(12, 5))
ax1, ax2 = fig.subplots(1, 2)
ax1.plot(ts, prey_net, label='ANN-based solution of prey')
ax1.plot(ts, prey_num, '.', label='numerical solution of prey')
ax1.plot(ts, pred_net, label='ANN-based solution of predator')
ax1.plot(ts, pred_num, '.', label='numerical solution of predator')
ax1.set_ylabel('population')
ax1.set_xlabel('t')
ax1.set_title('Comparing solutions')
ax1.legend()

ax2.set_title('Error of ANN solution from numerical solution')
ax2.plot(ts, prey_net-prey_num, label='error in prey number')
ax2.plot(ts, pred_net-pred_num, label='error in predator number')
ax2.set_ylabel('populator')
ax2.set_xlabel('t')
ax2.legend()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[17]:
```
<matplotlib.legend.Legend at 0x7fb359dec7f0>
```

2.3 Solving PDEs

Two-dimensional PDEs can be solved by the legacy `neurodiffeq.pde.solve2D` or the more flexible `neurodiffeq.solvers.Solver2D`.

Aain, just for the sake of notation in the following examples, here we see differentiation as an operation, then an PDE of $u(x, y)$ can be rewritten as:

$$F(u, x, y) = 0.$$

2.3.1 PDE Example 1: Laplace's Equation

Here we solve 2-D Laplace equation on a Cartesian boundary with Dirichlet boundary condition:

$$F(u, x, y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

for $(x, y) \in [0, 1] \times [0, 1]$

s.t.

$$u(x,y)\Big|_{x=0} = \sin(\pi y)$$
$$u(x,y)\Big|_{x=1} = 0$$
$$u(x,y)\Big|_{y=0} = 0$$
$$u(x,y)\Big|_{y=1} = 0$$

The analytical solution is

$$u(x,y) = \frac{\sin(\pi y)\sinh(\pi(1-x))}{\sinh(\pi)}$$

Here we have a Dirichlet boundary condition on both 4 edges of the orthogonal box. We will be using `DirichletBVP2D` for this boundary condition. The arguments `x_min_val`, `x_max_val`, `y_min_val` and `y_max_val` correspond to $u(x,y)\Big|_{x=0}$, $u(x,y)\Big|_{x=1}$, $u(x,y)\Big|_{y=0}$ and $u(x,y)\Big|_{y=1}$. Note that they should all be functions of x or y. These functions are expected to take in a `torch.tensor` and return a `torch.tensor`, so if the function involves some elementary functions like sin, we should use `torch.sin` instead of `numpy.sin`.

Like in the ODE case, we have two ways to solve 2-D PDEs.

1. The `neurodiffeq.pde.solve2D` function is almost the same as `solve` and `solve_system` in the `neurodiffeq.ode` module. Again, **this way is deprecated and won't be covered here**.
2. The `neurodiffeq.solvers.Solver2D` class is almost the same as `neurodiffeq.solvers.Solver1D`.

The difference is that we indicate the domain of our problem with `xy_min` and `xy_max`, they are tuples representing the 'lower left' point and 'upper right' point of our domain.

Also, we need to use `neurodiffeq.generators.Generator2D` and `neurodiffeq.monitors.Monitor2D`.

[18]:
```
from neurodiffeq.conditions import DirichletBVP2D
from neurodiffeq.solvers import Solver2D
from neurodiffeq.monitors import Monitor2D
from neurodiffeq.generators import Generator2D
import torch
```

[19]:
```
# Define the PDE system
# There's only one (Laplace) equation in the system, so the function maps (u, x, y)␣
↪to a single entry
laplace = lambda u, x, y: [
    diff(u, x, order=2) + diff(u, y, order=2)
]

# Define the boundary conditions
# There's only one function to be solved for, so we only have a single condition
conditions = [
    DirichletBVP2D(
        x_min=0, x_min_val=lambda y: torch.sin(np.pi*y),
        x_max=1, x_max_val=lambda y: 0,
        y_min=0, y_min_val=lambda x: 0,
```

(continues on next page)

(continued from previous page)

```
        y_max=1, y_max_val=lambda x: 0,
    )
]

# Define the neural network to be used
# Again, there's only one function to be solved for, so we only have a single network
nets = [
    FCNN(n_input_units=2, n_output_units=1, hidden_units=[512])
]

# Define the monitor callback
monitor=Monitor2D(check_every=10, xy_min=(0, 0), xy_max=(1, 1))
monitor_callback = monitor.to_callback()

# Instantiate the solver
solver = Solver2D(
    pde_system=laplace,
    conditions=conditions,
    xy_min=(0, 0),  # We can omit xy_min when both train_generator and valid_
↪generator are specified
    xy_max=(1, 1),  # We can omit xy_max when both train_generator and valid_
↪generator are specified
    nets=nets,
    train_generator=Generator2D((32, 32), (0, 0), (1, 1), method='equally-spaced-noisy
↪'),
    valid_generator=Generator2D((32, 32), (0, 0), (1, 1), method='equally-spaced'),
)

# Fit the neural network
solver.fit(max_epochs=200, callbacks=[monitor_callback])

# Obtain the solution
solution_neural_net_laplace = solver.get_solution()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Here we create a function to help us visualize the shape of the solution and the residual.

[20]:
```
from mpl_toolkits.mplot3d  import Axes3D
def plt_surf(xx, yy, zz, z_label='u', x_label='x', y_label='y', title=''):
    fig  = plt.figure(figsize=(16, 8))
    ax   = Axes3D(fig)
    surf = ax.plot_surface(xx, yy, zz, rstride=2, cstride=1, alpha=0.8, cmap='hot')
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    ax.set_zlabel(z_label)
    fig.suptitle(title)
    ax.set_proj_type('ortho')
    plt.show()
```

[21]:
```
xs, ys = np.linspace(0, 1, 101), np.linspace(0, 1, 101)
xx, yy = np.meshgrid(xs, ys)
sol_net = solution_neural_net_laplace(xx, yy, to_numpy=True)
plt_surf(xx, yy, sol_net, title='$u(x, y)$ as solved by neural network')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[22]:
```
solution_analytical_laplace = lambda x, y: np.sin(np.pi*y) * np.sinh(np.pi*(1-x))/np.
↪sinh(np.pi)
sol_ana = solution_analytical_laplace(xx, yy)
plt_surf(xx, yy, sol_net-sol_ana, z_label='residual', title='residual of the neural␣
↪network solution')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.3.2 PDE Example 2: 1-D Heat Equation

Here we solve 1-D heat equation:

$$\frac{\partial u}{\partial t} - k\frac{\partial^2 u}{\partial x^2} = 0$$

with an initial condition and 2 Neumann boundary on each end:

$$u(x,t)\Big|_{t=0} = \sin(\pi x)$$
$$\frac{\partial u(x,t)}{\partial x}\Big|_{x=0} = \pi \exp(-k\pi^2 t)$$
$$\frac{\partial u(x,t)}{\partial x}\Big|_{x=1} = -\pi \exp(-k\pi^2 t)$$

The analytical solution is:

$$u(x,t) = \sin(\pi\frac{x}{L})\exp(\frac{-k\pi^2 t}{L^2})$$

Since we are still solving in a 2-D space (x and t), we will still be using `solve2D`. We use a `IBVP1D` condition to enforce our initial and boundary condition. The arguments `t_min_val`, `x_min_prime`, and `x_max_prime` correspond to $u(x,t)\Big|_{t=0}$, $\frac{\partial u(x,t)}{\partial x}\Big|_{x=0}$ and $\frac{\partial u(x,t)}{\partial x}\Big|_{x=1}$.

[23]:
```
from neurodiffeq.conditions import IBVP1D
from neurodiffeq.pde import make_animation
```

[24]:
```
k, L, T = 0.3, 2, 3
# Define the PDE system
# There's only one (heat) equation in the system, so the function maps (u, x, y) to a␣
↪single entry
heat = lambda u, x, t: [
    diff(u, t) - k * diff(u, x, order=2)
]

# Define the initial and boundary conditions
# There's only one function to be solved for, so we only have a single condition␣
↪object
conditions = [
    IBVP1D(
```

(continues on next page)

(continued from previous page)

```
        t_min=0, t_min_val=lambda x: torch.sin(np.pi * x / L),
        x_min=0, x_min_prime=lambda t:  np.pi/L * torch.exp(-k*np.pi**2*t/L**2),
        x_max=L, x_max_prime=lambda t: -np.pi/L * torch.exp(-k*np.pi**2*t/L**2)
    )
]

# Define the neural network to be used
# Again, there's only one function to be solved for, so we only have a single network
nets = [
    FCNN(n_input_units=2, hidden_units=(32, 32))
]


# Define the monitor callback
monitor=Monitor2D(check_every=10, xy_min=(0, 0), xy_max=(L, T))
monitor_callback = monitor.to_callback()

# Instantiate the solver
solver = Solver2D(
    pde_system=heat,
    conditions=conditions,
    xy_min=(0, 0),  # We can omit xy_min when both train_generator and valid_
↪generator are specified
    xy_max=(L, T),  # We can omit xy_max when both train_generator and valid_
↪generator are specified
    nets=nets,
    train_generator=Generator2D((32, 32), (0, 0), (L, T), method='equally-spaced-noisy
↪'),
    valid_generator=Generator2D((32, 32), (0, 0), (L, T), method='equally-spaced'),
)

# Fit the neural network
solver.fit(max_epochs=200, callbacks=[monitor_callback])

# Obtain the solution
solution_neural_net_heat = solver.get_solution()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

We can use `make_animation` to animate the solution. Here we also check the residual of our solution.

[25]:
```
xs = np.linspace(0, L, 101)
ts = np.linspace(0, T, 101)
xx, tt = np.meshgrid(xs, ts)
make_animation(solution_neural_net_heat, xs, ts)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[25]:
```
<matplotlib.animation.FuncAnimation at 0x7fb3783f36d0>
```

[26]:
```
solution_analytical_heat = lambda x, t: np.sin(np.pi*x/L) * np.exp(-k * np.pi**2 * t /
↪ L**2)
sol_ana = solution_analytical_heat(xx, tt)
sol_net = solution_neural_net_heat(xx, tt, to_numpy=True)
plt_surf(xx, tt, sol_net-sol_ana, y_label='t', z_label='residual of the neural␣
↪network solution')
```

(continues on next page)

(continued from previous page)

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.4 Irregular Domain

[27]:
```
import numpy as np
import torch
from torch import nn
from torch import optim
from neurodiffeq import diff
from neurodiffeq.networks import FCNN
from neurodiffeq.pde import solve2D, Monitor2D
from neurodiffeq.generators import Generator2D, PredefinedGenerator
from neurodiffeq.pde import CustomBoundaryCondition, Point, DirichletControlPoint
import matplotlib.pyplot as plt
import matplotlib.tri as tri
```

`neurodiffeq` also implemented a method (https://ieeexplore.ieee.org/document/5061501) to impose Dirichlet boundary condition on an irregular domain. Here we show a problem that is defined on a star shaped domain. The following cell are some helper functions we will use later.

[28]:
```
def get_grid(x_from_to, y_from_to, x_n_points=100, y_n_points=100, as_tensor=False):
    x_from, x_to = x_from_to
    y_from, y_to = y_from_to
    if as_tensor:
        x = torch.linspace(x_from, x_to, x_n_points)
        y = torch.linspace(y_from, y_to, y_n_points)
        return torch.meshgrid(x, y)
    else:
        x = np.linspace(x_from, x_to, x_n_points)
        y = np.linspace(y_from, y_to, y_n_points)
        return np.meshgrid(x, y)

def create_contour(ax, xs, ys, zs, cdbc=None):
    triang = tri.Triangulation(xs, ys)
    xs = xs[triang.triangles].mean(axis=1)
    ys = ys[triang.triangles].mean(axis=1)
    if cdbc:
        xs, ys = torch.tensor(xs), torch.tensor(ys)
        in_domain = cdbc.in_domain(xs, ys).detach().numpy()
        triang.set_mask(~in_domain)

    contour = ax.tricontourf(triang, zs, cmap='coolwarm')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_aspect('equal', adjustable='box')
    return contour

def compare_contour(sol_net, sol_ana, eval_on_xs, eval_on_ys, cdbc=None):
    eval_on_xs, eval_on_ys = eval_on_xs.flatten(), eval_on_ys.flatten()
    s_net = sol_net(eval_on_xs, eval_on_ys, to_numpy=True)
    s_ana = sol_ana(eval_on_xs, eval_on_ys)
```

(continues on next page)

(continued from previous page)

```
    fig = plt.figure(figsize=(18, 4))

    ax1 = fig.add_subplot(131)
    cs1 = create_contour(ax1, eval_on_xs, eval_on_ys, s_net, cdbc)
    ax1.set_title('ANN-based solution')
    cbar1 = fig.colorbar(cs1, format='%.0e', ax=ax1)

    ax2 = fig.add_subplot(132)
    cs2 = create_contour(ax2, eval_on_xs, eval_on_ys, s_ana, cdbc)
    ax2.set_title('analytical solution')
    cbar2 = fig.colorbar(cs2, format='%.0e', ax=ax2)

    ax3 = fig.add_subplot(133)
    cs3 = create_contour(ax3, eval_on_xs, eval_on_ys, s_net-s_ana, cdbc)
    ax3.set_title('residual of ANN-based solution')
    cbar3 = fig.colorbar(cs3, format='%.0e', ax=ax3)
```

The problem we want to solve is defined on a hexagram that is centered at the origin. The uppermost vertex of the hexagram locates at $(0, 1)$ and the lowermost vertex locates at $(0, -1)$. The differential equation is:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + e^u = 1 + x^2 + y^2 + \frac{4}{(1 + x^2 + y^2)^2}$$

We have a Dirichlet condition along the boundary of the domain:

$$u(x, y) = \ln(1 + x^2 + y^2)$$

The analytical solution is:

$$u(x, y) = \ln(1 + x^2 + y^2)$$

[29]:
```
def solution_analytical_star(x, y):
    return np.log(1+x**2+y**2)
```

`neurodiffeq.pde.CustomBoundaryCondition` imposes boundary condition on a irregular domain. To specify this domain, we will pass to `CustomBoundaryCondition` a collection of control points that fall on the boundary. Here on each edge of the hexagram, we create 11 `neurodiffeq.pde.DirichletControlPoint` that impose the value of u at these location. This collection of `DirichletControlPoint` are then passed to `CustomBoundaryCondition` to fit the trial solution.

[30]:
```
edge_length = 2.0 / np.sin(np.pi/3) / 4
points_on_each_edge = 11
step_size = edge_length / (points_on_each_edge-1)

direction_theta  =  np.pi*2/3
left_turn_theta  =  np.pi*1/3
right_turn_theta = -np.pi*2/3

control_points_star = []
point_x, point_y = 0.0, -1.0
for i_edge in range(12):
    for i_step in range(points_on_each_edge-1):
```

(continues on next page)

(continued from previous page)

```
        control_points_star.append(
            DirichletControlPoint(
                loc=(point_x, point_y),
                val=solution_analytical_star(point_x, point_y)
            )
        )
        point_x += step_size*np.cos(direction_theta)
        point_y += step_size*np.sin(direction_theta)
    direction_theta += left_turn_theta if (i_edge % 2 == 0) else right_turn_theta

cdbc_star = CustomBoundaryCondition(
    center_point=Point((0.0, 0.0)),
    dirichlet_control_points=control_points_star
)
```

Here we use a set of predifined points as our training set. These points are from a 32×32 grid in $(-1, 1) \times (-1, 1)$. We filter out the points that don't fall in the domain (`CustomBoundaryCondition` has an `in_domain` method that returns a mask of points that are in the domain) and use the rest as the training set. We also specify a validation set that is denser.

[31]:
```
%matplotlib notebook
def to_np(tensor):
    return tensor.detach().numpy()

xx_train, yy_train = get_grid(
    x_from_to=(-1, 1), y_from_to=(-1, 1),
    x_n_points=32, y_n_points=32,
    as_tensor=True
)
is_in_domain_train = cdbc_star.in_domain(xx_train, yy_train)
xx_train, yy_train = to_np(xx_train), to_np(yy_train)
xx_train, yy_train = xx_train[is_in_domain_train], yy_train[is_in_domain_train]
train_gen = PredefinedGenerator(xx_train, yy_train)


xx_valid, yy_valid = get_grid(
    x_from_to=(-1, 1), y_from_to=(-1, 1),
    x_n_points=100, y_n_points=100,
    as_tensor=True
)
is_in_domain_valid = cdbc_star.in_domain(xx_valid, yy_valid)
xx_valid, yy_valid = to_np(xx_valid), to_np(yy_valid)
xx_valid, yy_valid = xx_valid[is_in_domain_valid], yy_valid[is_in_domain_valid]
valid_gen = PredefinedGenerator(xx_valid, yy_valid)

plt.figure(figsize=(7, 7))
plt.scatter(xx_train, yy_train)
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.gca().set_aspect('equal', adjustable='box')
plt.title('training points');
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

In this case, since we know the analytical solution. We can keep track of the mean squared error of our approximation. This can be done by passing a dictionary to the `metrics` keyword of the `solve2D` function. The (key, value) are

(metric name, function that calculates the metric).

[32]:
```
%matplotlib notebook
def mse(u, x, y):
    true_u = torch.log(1+x**2+y**2)
    return torch.mean( (u - true_u)**2 )

# Define the differential equation
def de_star(u, x, y):
    return [diff(u, x, order=2) + diff(u, y, order=2) + torch.exp(u) - 1.0 - x**2 -␣
↪y**2 - 4.0/(1.0+x**2+y**2)**2]

# fully connected network with one hidden layer (40 hidden units with Sigmoid␣
↪activation)
net = FCNN(n_input_units=2, hidden_units=(40, 40), actv=nn.ELU)
adam = optim.Adam(params=net.parameters(), lr=0.01)

# Define the monitor callback
monitor = Monitor2D(check_every=10, xy_min=(-1, -1), xy_max=(1, 1))
monitor_callback = monitor.to_callback()

# Instantiate the solver
solver = Solver2D(
    pde_system=de_star,
    conditions=[cdbc_star],
    xy_min=(-1, -1),  # We can omit xy_min when both train_generator and valid_
↪generator are specified
    xy_max=(1, 1),    # We can omit xy_max when both train_generator and valid_
↪generator are specified
    nets=nets,
    train_generator=train_gen,
    valid_generator=valid_gen,
)

# Fit the neural network, train on 32 x 32 grids
solver.fit(max_epochs=100, callbacks=[monitor_callback])

# Obtain the solution
solution_neural_net_star = solver.get_solution()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

We can compare the neural network solution with the analytical solution:

[34]:
```
%matplotlib notebook
compare_contour(
    solution_neural_net_star,
    solution_analytical_star,
    xx_valid, yy_valid, cdbc=cdbc_star
)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[ ]:

CHAPTER 3

Advanced Uses

[1]:
```
import numpy as np
import torch
from neurodiffeq.neurodiffeq import safe_diff as diff
from neurodiffeq.ode import solve, solve_system
from neurodiffeq.solvers import Solver1D
from neurodiffeq.monitors import Monitor1D
from neurodiffeq.conditions import IVP

import matplotlib.pyplot as plt
%matplotlib notebook
```

3.1 Tuning the Solver

The `solve*` functions (in `neurodiffeq.ode`, `neurodiffeq.pde`, `neurodiffeq.pde_spherical`) and `Solver*` classes (in `neurodiffeq.solvers`) choose some hyperparameters by default. For example, in `neurodiffeq.solver.Solver1D`, by default: * the solution is approximated by a fully connected network of 2 hidden layers with 32 units each (tanh activation), * for each epoch we train on 16 different points that are generated by adding a Gaussian noise on the 32 equally spaced points on the t domain, * an Adam optimizer with learning rate 0.001 is used

Sometimes we may want to choose these hyperparameters ourselves. We will be using the harmonic oscillator problem from above to demonstrate how to do that.

3.1.1 Simple Harmonic Oscillator Example

In the following example, we demonstrate how to change these default settings using the harmonic oscillator as an example.

The differential equation and the initial condition are:

$$\frac{\partial^2 u}{\partial t^2} + u = 0$$
$$u\Big|_{t=0} = 0 \quad \frac{\partial u}{\partial t}\Big|_{t=0} = 1$$

[2]:
```
# Note that the function maps (u, t) to a single-entry list
harmonic_oscillator = lambda u, t: [ diff(u, t, order=2) + u ]
init_val_ho = IVP(t_0=0.0, u_0=0.0, u_0_prime=1.0)
```

3.1.2 Specifying the Networks

[3]:
```
from neurodiffeq.networks import FCNN # fully-connect neural network
import torch.nn as nn                 # PyTorch neural network module
```

Whether you are using a `neurodiffeq.solvers.Solver1D` instance or the legacy functions `neurodiffeq.ode.solve` and `neurodiffeq.ode.solve_system` to solve differential equations, you can specify the network architecture you want to use.

The architecture must be defined as a subclass of `torch.nn.Module`. If you are familiar with PyTorch, this process couldn't be simpler. If you don't know PyTorch at all, we have defined a `neurodiffeq.networks.FCNN` for you. `FCNN` stands for Fully-Connected Neural Network. You can tweak it any how you want by specifying

1. `hidden_units`: number of units for each hidden layer. If you have 3 hidden layers with 32, 64, and 16 neurons respectively, then `hidden_units` should be a tuple `(32, 64, 16)`.
2. `actv`: a `torch.nn.Module` *class*. e.g. `nn.Tanh`, `nn.Sigmoid`. Impirically, Swish works better in many situations. We have implemented a `Swish` activation in `neurodiffeq.networks` for you to try out.
3. `n_input_units` and `n_output_units`: number of input/output units of the network. This is largely dependent on your problem. In most cases, `n_output_units` should be 1. And `n_input_units` should be the number of independent variables. In the case of ODE, this is 1, since we only have a single independent variable t.

If you want more flexibility than only using fully connected networks, check out PyTorch's tutorials on defining your custom `torch.nn.Module`. **Pro tip: it's simpler than you think :)**

Once you figure out how to define your own network (as an instance of `torch.nn.Module`), you can pass it to

1. `neurodiffeq.solvers.Solver1D` and other `Solvers` in this Module by specifying `nets=[your_net1, your_net2, ...]`; or
2. `neurodiffeq.ode.solve`, `neurodiffeq.pde.solve2D`, `neurodiffeq.pde_spherical.solve_spherical`, etc., by specifying `net=your_net`; or
3. `neurodiffeq.ode.solve_system`, `neurodiffeq.pde.solve2D_sytem`, `neurodiffeq.pde_spherical.solve_spherical_system`, etc., by specifying `nets=[your_net1, your_net2, ...]`.

Notes: * Only the 1st way (using a `Solver`) is recommended, the 2nd and 3rd way (using a `solve*` function) are deprecated will some day be removed; * In the 2nd case, these functions assumes you only solving a single equation for a single function, so you pass in **a single network** `net=...`; * In the 1st and 3rd cases, they assume you are solving arbitraily many equations for arbitrarily functions, so you pass in **a list of networks** `nets=[...]`.

Here we create a fully connected network with 3 hidden layers, each with 16 units and tanh activation. We then use it to fit our ODE solution.

[4]:
```
%matplotlib notebook
# Specify the network architecture
net_ho = FCNN(
    hidden_units=(16, 16, 16), actv=nn.Tanh
)

# Create a monitor callback
from neurodiffeq.monitors import Monitor1D
monitor_callback = Monitor1D(t_min=0.0, t_max=2*np.pi, check_every=100).to_callback()

# Create a solver
solver = Solver1D(
    ode_system=harmonic_oscillator,  # Note that `harmonic_oscillator` returns a␣
↪single-entry list
    conditions=[init_val_ho],        # Again, `conditions` is a single-entry list
    t_min=0.0,
    t_max=2*np.pi,
    nets=[net_ho],                   # Again, `nets` is a single-entry list
)

# Fit the solver
solver.fit(max_epochs=1000, callbacks=[monitor_callback])

# Obtain the solution
solution_ho = solver.get_solution()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

3.1.3 Specifying the Training Set and Validation Set

Both `Solver*` classes and `solve*` functions train the neural network on a new set of points, randomly sampled every time. These examples are ts drawn from the domain of t. The way these ts are generated can be specified by passing a `neurodiffeq.generators.BaseGenerator` object as the `train_generator` argument (and `valid_generator` argument) to `Solver*` classes or `solve*` functions. An `Generator` can be intialized by the following arguments:

- `size`: the number of ts generated for each epoch
- `t_min` and `t_max`: the domain of t from which we want to draw ts
- `method`: a string indicating how to generate the ts. It should be one of the following: 'uniform', 'equally-spaced', 'equally-spaced-noisy'. If 'uniform', each t will be drawn independently from the uniform distribution Unif(`t_min`, `t_max`). If 'equally-spaced', all ts generated in the same epoch will form a grid where each t is equally spaced. 'equally-spaced-noisy' is a noisy version of 'equally-spaced' where we add a Gaussian noise $\epsilon \sim \mathcal{N}(0, (\texttt{t\_max-t\_min})/(4*\texttt{size}))$

Here we create an `Generator` that generates 64 ts drawn from a uniform distribution for every epoch. Then we use it to solve the ODE. In the meantime, foor every epoch, we will use another `Generator` that generates 128 ts that are equally spaced in the domain we want to solve.

[5]:
```
from neurodiffeq.generators import Generator1D
```

[6]:
```
%matplotlib notebook
# specify the training set and validation set
train_gen = Generator1D(size=64,  t_min=0.0, t_max=2*np.pi, method='uniform')
```

(continues on next page)

(continued from previous page)

```
valid_gen = Generator1D(size=128, t_min=0.0, t_max=2*np.pi, method='equally-spaced')

# solve the ODE
solver = Solver1D(
    ode_system=harmonic_oscillator,
    conditions=[init_val_ho],
    t_min=0.0,
    t_max=2*np.pi,
    train_generator=train_gen,
    valid_generator=valid_gen,
)
solver.fit(
    max_epochs=1000,
    callbacks=[Monitor1D(t_min=0.0, t_max=2*np.pi, check_every=100).to_callback()]
)

solution_ho = solver.get_solution()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

3.1.4 Specifying the Optimizer

We can change the optimization algorithms by passing a `torch.optim.Optimizer` object to `Solver*` classes and `solve*` functions as the `optimizer` argument.

If you are familiar with PyTorch, you know that to initialize an `Optimizer`, we need to tell it the parameters to optimize. In other words, if we want to use a different optimizer from the default one, we also need to create our own networks.

Here we create a fully connected network and an `SGD` optimizer to optimize its weights. Then we use them to solve the ODE.

[7]:
```
from torch.optim import SGD
```

[8]:
```
%matplotlib notebook
# specify the network architecture
net_ho = FCNN(
    n_input_units=1,
    n_output_units=1,
    hidden_units=(16, 16, 16),
    actv=nn.Tanh,
)

nets = [net_ho]

# specify the optimizer
from itertools import chain

sgd_ho = SGD(
    chain.from_iterable(n.parameters() for n in nets), # this gives all parameters in␣
↪`nets`
    lr=0.001,                                          # learning rate
    momentum=0.99,                                     # momentum of SGD
)
```

(continues on next page)

(continued from previous page)

```
# solve the ODE
solver = Solver1D(
    ode_system=harmonic_oscillator,
    conditions=[init_val_ho],
    t_min=0.0,
    t_max=2*np.pi,
    nets=nets,
    optimizer=sgd_ho,
)

solver.fit(
    max_epochs=1000,
    callbacks=[Monitor1D(t_min=0.0, t_max=2*np.pi, check_every=100).to_callback()]
)

solution_ho = solver.get_solution()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

3.1.5 Specifying the Loss Function

We can change the loss function by passing a `torch.nn._Loss` object to `solve` and `solve_system` as the `criterion` argument.

Here we use the mean absolute loss to solve the ODE.

[9]:
```
from torch.nn import L1Loss
```

[10]:
```
%matplotlib notebook
# solve the ODE
solver = Solver1D(
    ode_system=harmonic_oscillator,
    conditions=[init_val_ho],
    t_min=0.0,
    t_max=2*np.pi,
    criterion=L1Loss(),
)
solver.fit(
    max_epochs=1000,
    callbacks=[Monitor1D(t_min=0.0, t_max=2*np.pi, check_every=100).to_callback()]
)

solution_ho = solver.get_solution()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

3.2 Access the Internals

When the network, generator, optimizer and loss function are specified outside `solve` and `solve_system` function, users will naturally have access to these objects. We may still want to access these objects when we are using

default network architecture, generator, optimizer and loss function. We can get these internal objects by setting the `return_internal` keyword to `True`. This will add a third element in the returned tuple, which is a dictionary containing the reference to the network, example generator, optimizer and loss function.

3.2.1 Using a `solve*` function to get internals

[11]:
```
# specify the ODE system
parametric_circle = lambda x1, x2, t : [diff(x1, t) - x2,
                                        diff(x2, t) + x1]
# specify the initial conditions
init_vals_pc = [
    IVP(t_0=0.0, u_0=0.0),
    IVP(t_0=0.0, u_0=1.0),
]

# solve the ODE system
solution_pc, _, internal = solve_system(
    ode_system=parametric_circle,
    conditions=init_vals_pc,
    t_min=0.0, t_max=2*np.pi,
    return_internal=True
)
```

```
/Users/liushuheng/Documents/GitHub/neurodiffeq/neurodiffeq/ode.py:260: FutureWarning:␣
↪The `solve_system` function is deprecated, use a `neurodiffeq.solvers.Solver1D`␣
↪instance instead
  warnings.warn(
```

[12]:
```
internal
```

[12]:
```
{'nets': [FCNN(
    (NN): Sequential(
      (0): Linear(in_features=1, out_features=32, bias=True)
      (1): Tanh()
      (2): Linear(in_features=32, out_features=32, bias=True)
      (3): Tanh()
      (4): Linear(in_features=32, out_features=2, bias=True)
    )
  ),
  FCNN(
    (NN): Sequential(
      (0): Linear(in_features=1, out_features=32, bias=True)
      (1): Tanh()
      (2): Linear(in_features=32, out_features=32, bias=True)
      (3): Tanh()
      (4): Linear(in_features=32, out_features=2, bias=True)
    )
  )],
 'conditions': [<neurodiffeq.conditions.IVP at 0x7fd0e08d67c0>,
  <neurodiffeq.conditions.IVP at 0x7fd0e08d6730>],
 'train_generator': SamplerGenerator(size=32, generator=Generator1D(size=32, t_min=0.
↪0, t_max=6.283185307179586, method='equally-spaced-noisy', noise_std=0.
↪04908738521234052)),
 'valid_generator': SamplerGenerator(size=32, generator=Generator1D(size=32, t_min=0.
↪0, t_max=6.283185307179586, method='equally-spaced', noise_std=0.
↪04908738521234052)),
 'optimizer': Adam (
```

(continues on next page)

(continued from previous page)

```
 Parameter Group 0
     amsgrad: False
     betas: (0.9, 0.999)
     eps: 1e-08
     lr: 0.001
     weight_decay: 0
 ),
 'criterion': <function neurodiffeq.solvers.BaseSolver.__init__.<locals>.<lambda>(r)>}
```

3.2.2 Using a `Solver*` instance to get internals

You get more internal objects when using the `Solvers`. The process is demonstrated as follows:

[13]:
```
parametric_circle = lambda x1, x2, t: [diff(x1, t) - x2, diff(x2, t) + x1]

init_vals_pc = [
    IVP(t_0=0.0, u_0=0.0),
    IVP(t_0=0.0, u_0=1.0),
]

solver = Solver1D(
    ode_system=parametric_circle,
    conditions=init_vals_pc,
    t_min=0.0,
    t_max=2*np.pi,
)

solver.fit(max_epochs=100)
internals = solver.get_internals()
```

[14]:
```
internals
```

[14]:
```
{'metrics': {},
 'n_batches': {'train': 1, 'valid': 4},
 'best_nets': [FCNN(
    (NN): Sequential(
      (0): Linear(in_features=1, out_features=32, bias=True)
      (1): Tanh()
      (2): Linear(in_features=32, out_features=32, bias=True)
      (3): Tanh()
      (4): Linear(in_features=32, out_features=1, bias=True)
    )
  ),
  FCNN(
    (NN): Sequential(
      (0): Linear(in_features=1, out_features=32, bias=True)
      (1): Tanh()
      (2): Linear(in_features=32, out_features=32, bias=True)
      (3): Tanh()
      (4): Linear(in_features=32, out_features=1, bias=True)
    )
  )],
 'criterion': <function neurodiffeq.solvers.BaseSolver.__init__.<locals>.<lambda>(r)>,
 'conditions': [<neurodiffeq.conditions.IVP at 0x7fd0e08d6430>,
  <neurodiffeq.conditions.IVP at 0x7fd0e08d6340>],
```

(continues on next page)

(continued from previous page)

```
'global_epoch': 100,
'lowest_loss': 0.023892831479509158,
'n_funcs': 2,
'nets': [FCNN(
   (NN): Sequential(
     (0): Linear(in_features=1, out_features=32, bias=True)
     (1): Tanh()
     (2): Linear(in_features=32, out_features=32, bias=True)
     (3): Tanh()
     (4): Linear(in_features=32, out_features=1, bias=True)
   )
 ),
 FCNN(
   (NN): Sequential(
     (0): Linear(in_features=1, out_features=32, bias=True)
     (1): Tanh()
     (2): Linear(in_features=32, out_features=32, bias=True)
     (3): Tanh()
     (4): Linear(in_features=32, out_features=1, bias=True)
   )
 )],
'optimizer': Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    eps: 1e-08
    lr: 0.001
    weight_decay: 0
),
'diff_eqs': <function __main__.<lambda>(x1, x2, t)>,
'generator': {'train': SamplerGenerator(size=32, generator=Generator1D(size=32, t_
↪min=0.0, t_max=6.283185307179586, method='equally-spaced-noisy', noise_std=0.
↪04908738521234052)),
 'valid': SamplerGenerator(size=32, generator=Generator1D(size=32, t_min=0.0, t_
↪max=6.283185307179586, method='equally-spaced', noise_std=0.04908738521234052))},
'train_generator': SamplerGenerator(size=32, generator=Generator1D(size=32, t_min=0.
↪0, t_max=6.283185307179586, method='equally-spaced-noisy', noise_std=0.
↪04908738521234052)),
'valid_generator': SamplerGenerator(size=32, generator=Generator1D(size=32, t_min=0.
↪0, t_max=6.283185307179586, method='equally-spaced', noise_std=0.
↪04908738521234052)),
't_min': 0.0,
't_max': 6.283185307179586}
```

[ ]:

CHAPTER 4

API Reference

4.1 *neurodiffeq.neurodiffeq*

neurodiffeq.neurodiffeq.**diff**(*u*, *t*, *order=1*, *shape_check=True*)

The derivative of a variable with respect to another. `diff` defaults to the behaviour of `safe_diff`.

Parameters

- **u** (*torch.Tensor*) – The u in $\frac{\partial u}{\partial t}$.
- **t** (*torch.Tensor*) – The t in $\frac{\partial u}{\partial t}$.
- **order** (*int*) – The order of the derivative, defaults to 1.
- **shape_check** (*bool*) – Whether to perform shape checking or not, defaults to True (since v0.2.0).

Returns The derivative evaluated at t.

Return type *torch.Tensor*

neurodiffeq.neurodiffeq.**safe_diff**(*u*, *t*, *order=1*)

The derivative of a variable with respect to another. Both tensors must have a shape of (n_samples, 1) See this issue comment for details

Parameters

- **u** (*torch.Tensor*) – The u in $\frac{\partial u}{\partial t}$.
- **t** (*torch.Tensor*) – The t in $\frac{\partial u}{\partial t}$.
- **order** (*int*) – The order of the derivative, defaults to 1.

Returns The derivative evaluated at `t`.

Return type *torch.Tensor*

neurodiffeq.neurodiffeq.**unsafe_diff**(*u*, *t*, *order=1*)

The derivative of a variable with respect to another. While there's no requirement for shapes, errors could occur in some cases. See this issue for details

Parameters

- **u** (*torch.Tensor*) – The u in $\frac{\partial u}{\partial t}$.
- **t** (*torch.Tensor*) – The t in $\frac{\partial u}{\partial t}$.
- **order** (*int*) – The order of the derivative, defaults to 1.

Returns The derivative evaluated at `t`.

Return type *torch.Tensor*

4.2 neurodiffeq.networks

4.3 neurodiffeq.conditions

class neurodiffeq.conditions.**BaseCondition**

Bases: `object`

Base class for all conditions.

A condition is a tool to *re-parameterize* the output(s) of a neural network. such that the re-parameterized output(s) will automatically satisfy initial conditions (ICs) and boundary conditions (BCs) of the PDEs/ODEs that are being solved.

Note:

- The nouns *(re-)parameterization* and *condition* are used interchangeably in the documentation and library.
- The verbs *(re-)parameterize* and *enforce* are different in that:
  - *(re)parameterize* is said of network outputs;
  - *enforce* is said of networks themselves.

---

enforce(*net*, **coordinates*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, **input_tensors*)

Re-parameterizes output(s) of a network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.

- **input_tensors** (*torch.Tensor*) – Inputs to the neural network; i.e., sampled coordinates; i.e., independent variables.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

Note: This method is **abstract** for BaseCondition

---

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.conditions.**BundleIVP**(*t_0=None*, *u_0=None*, *u_0_prime=None*, *bundle_conditions={}*)

Bases: *neurodiffeq.conditions.BaseCondition*

An initial value problem of one of the following forms:

- Dirichlet condition: $u(t_0, \boldsymbol{\theta}) = u_0$.
- Neumann condition: $\left.\dfrac{\partial u}{\partial t}\right|_{t=t_0}(\boldsymbol{\theta}) = u_0'$.

Here $\boldsymbol{\theta} = (\theta_1, \theta_2, ..., \theta_n) \in \mathbb{R}^n$, where each θ_i represents a parameter, or a condition, of the ODE system that we want to solve.

Parameters

- **t_0** (*float*) – The initial time.
- **u_0** (*float*) – The initial value of u. $u(t_0, \boldsymbol{\theta}) = u_0$.
- **u_0_prime** (*float, optional*) – The initial derivative of u w.r.t. t. $\left.\dfrac{\partial u}{\partial t}\right|_{t=t_0}(\boldsymbol{\theta}) = u_0'$. Defaults to None.
- **bundle_conditions** (*dict{str: int, .., str: int}*) – The initial conditions that will be included in the total bundle, in addition to the parameters of the ODE system. The values asociated with their respective keys used in bundle_conditions (e.g bundle_conditions={'t_0': 0, 'u_0': 1, 'u_0_prime': 2}), must reflect the index of the tuple used in theta_min and theta_max in `neurodiffeq.solvers.BundleSolver1D`, (e.g theta_min=(t_0_min, u_0_min, u_0_prime_min)). Defaults to {}

enforce(*net*, **coordinates*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, *t*, **theta*)

Re-parameterizes outputs such that the Dirichlet/Neumann condition is satisfied.

if t_0 is not included in the bundle:

- For Dirichlet condition, the re-parameterization is $u(t, \boldsymbol{\theta}) = u_0 + \left(1 - e^{-(t-t_0)}\right) \text{ANN}(t, \boldsymbol{\theta})$
- For Neumann condition, the re-parameterization is $u(t, \boldsymbol{\theta}) = u_0 + (t - t_0)u_0' + \left(1 - e^{-(t-t_0)}\right)^2 \text{ANN}(t, \boldsymbol{\theta})$

if t_0 is included in the bundle:

- For Dirichlet condition, the re-parameterization is $u(t, \boldsymbol{\theta}) = u_0 + (t - t_0) \text{ANN}(t, \boldsymbol{\theta})$
- For Neumann condition, the re-parameterization is $u(t, \boldsymbol{\theta}) = u_0 + (t - t_0)u_0' + (t - t_0)^2 \text{ANN}(t, \boldsymbol{\theta})$

Where ANN is the neural network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **t** (*torch.Tensor*) – First input to the neural network; i.e., sampled time-points; i.e., independent variables.
- **theta** (*tuple[torch.Tensor, .., torch.Tensor]*) – Rest of the inputs to the neural network; i.e., sampled bundle-points

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.conditions.**DirichletBVP**(*t_0*, *u_0*, *t_1*, *u_1*)

Bases: *neurodiffeq.conditions.BaseCondition*

A double-ended Dirichlet boundary condition: $u(t_0) = u_0$ and $u(t_1) = u_1$.

Parameters

- **t_0** (*float*) – The initial time.
- **u_0** (*float*) – The initial value of u. $u(t_0) = u_0$.
- **t_1** (*float*) – The final time.
- **u_1** (*float*) – The initial value of u. $u(t_1) = u_1$.

enforce(*net*, **coordinates*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, *t*)

Re-parameterizes outputs such that the Dirichlet condition is satisfied on both ends of the domain.

The re-parameterization is $u(t) = (1-\tilde{t})u_0 + \tilde{t}u_1 + \left(1 - e^{(1-\tilde{t})\tilde{t}}\right)\text{ANN}(t)$, where $\tilde{t} = \dfrac{t-t_0}{t_1-t_0}$ and ANN is the neural network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **t** (*torch.Tensor*) – Input to the neural network; i.e., sampled time-points or another independent variable.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.conditions.**DirichletBVP2D**(*x_min*, *x_min_val*, *x_max*, *x_max_val*, *y_min*, *y_min_val*, *y_max*, *y_max_val*)

Bases: *neurodiffeq.conditions.BaseCondition*

An Dirichlet boundary condition on the boundary of $[x_0, x_1] \times [y_0, y_1]$, where

- $u(x_0, y) = f_0(y)$;
- $u(x_1, y) = f_1(y)$;
- $u(x, y_0) = g_0(x)$;
- $u(x, y_1) = g_1(x)$.

Parameters

- **x_min** (*float*) – The lower bound of x, the x_0.
- **x_min_val** (*callable*) – The boundary value on $x = x_0$, i.e. $f_0(y)$.
- **x_max** (*float*) – The upper bound of x, the x_1.
- **x_max_val** (*callable*) – The boundary value on $x = x_1$, i.e. $f_1(y)$.
- **y_min** (*float*) – The lower bound of y, the y_0.
- **y_min_val** (*callable*) – The boundary value on $y = y_0$, i.e. $g_0(x)$.
- **y_max** (*float*) – The upper bound of y, the y_1.

- **y_max_val** (*callable*) – The boundary value on $y = y_1$, i.e. $g_1(x)$.

enforce(*net*, **coordinates*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, *x*, *y*)

Re-parameterizes outputs such that the Dirichlet condition is satisfied on all four sides of the domain.

The re-parameterization is $u(x,y) = A(x,y) + \tilde{x}(1-\tilde{x})\tilde{y}(1-\tilde{y})\text{ANN}(x,y)$, where

$$\tilde{x} = \frac{x - x_0}{x_1 - x_0},$$

$$\tilde{y} = \frac{y - y_0}{y_1 - y_0},$$

and ANN is the neural network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **x** (*torch.Tensor*) – x-coordinates of inputs to the neural network; i.e., the sampled x-coordinates.
- **y** (*torch.Tensor*) – y-coordinates of inputs to the neural network; i.e., the sampled y-coordinates.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.conditions.**DirichletBVPSpherical**(*r_0*, *f*, *r_1=None*, *g=None*)

Bases: *neurodiffeq.conditions.BaseCondition*

The Dirichlet boundary condition for the interior and exterior boundary of the sphere, where the interior boundary is not necessarily a point. The conditions are:

- $u(r_0, \theta, \phi) = f(\theta, \phi)$
- $u(r_1, \theta, \phi) = g(\theta, \phi)$

Parameters

- **r_0** (*float*) – The radius of the interior boundary. When $r_0 = 0$, the interior boundary collapses to a single point (center of the ball).
- **f** (*callable*) – The value of u on the interior boundary. $u(r_0, \theta, \phi) = f(\theta, \phi)$.
- **r_1** (*float or None*) – The radius of the exterior boundary. If set to None, *g* must also be None.
- **g** (*callable or None*) – The value of u on the exterior boundary. $u(r_1, \theta, \phi) = g(\theta, \phi)$. If set to None, *r_1* must also be set to None.

enforce(*net*, **coordinates*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, *r*, *theta*, *phi*)

Re-parameterizes outputs such that the Dirichlet condition is satisfied on both spherical boundaries.

- If both inner and outer boundaries are specified $u(r_0, \theta, \phi) = f(\theta, \phi)$ and $u(r_1, \theta, \phi) = g(\theta, \phi)$:

  The re-parameterization is $\left(1-\tilde{r}\right) f(\theta, \phi) + \tilde{r} g(\theta, \phi) + \left(1 - e^{\tilde{r}(1-\tilde{r})}\right) \mathrm{ANN}(r, \theta, \phi)$ where $\tilde{r} = \dfrac{r - r_0}{r_1 - r_0}$;

- If only one boundary is specified (inner or outer) $u(r_0, \theta, \phi) = f(\theta, \phi)$

  The re-parameterization is $f(\theta, \phi) + \left(1 - e^{-|r - r_0|}\right) \mathrm{ANN}(r, \theta, \phi)$;

where ANN is the neural network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **r** (*torch.Tensor*) – The radii (or r-component) of the inputs to the network.
- **theta** (*torch.Tensor*) – The co-latitudes (or θ-component) of the inputs to the network.
- **phi** (*torch.Tensor*) – The longitudes (or ϕ-component) of the inputs to the network.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.conditions.**DirichletBVPSphericalBasis**(*r_0*, *R_0*, *r_1=None*, *R_1=None*, *max_degree=None*)

Bases: *neurodiffeq.conditions.BaseCondition*

Similar to `neurodiffeq.conditions.DirichletBVPSpherical`. The only difference is this condition is enforced on a neural net that only takes in r and returns the spherical harmonic coefficients R(r). We constrain the coefficients $R_k(r)$ in $u(r,\theta,\phi) = \sum_k R_k(r) Y_k(\theta,\phi)$, where $\{Y_k(\theta,\phi)\}_{k=1}^{K}$ can be **any spherical function basis**. A recommended choice is the real spherical harmonics $Y_l^m(\theta,\phi)$, where l is the degree of the spherical harmonics and m is the order of the spherical harmonics.

The boundary conditions are: $\mathbf{R}(r_0) = \mathbf{R}_0$ and $\mathbf{R}(r_1) = \mathbf{R}_1$, where $\mathbf{R}$ is a vector whose components are $\{R_k\}_{k=1}^{K}$

Parameters

- **r_0** (*float*) – The radius of the interior boundary. When r_0 = 0, the interior boundary is collapsed to a single point (center of the ball).
- **R_0** (*torch.Tensor*) – The value of harmonic coefficients $\mathbf{R}$ on the interior boundary. $\mathbf{R}(r_0) = \mathbf{R}_0$.
- **r_1** (*float or None*) – The radius of the exterior boundary. If set to None, *R_1* must also be None
- **R_1** (*torch.Tensor*) – The value of harmonic coefficients $\mathbf{R}$ on the exterior boundary. $\mathbf{R}(r_1) = \mathbf{R}_1$.
- **max_degree** (*int*) – **[DEPRECATED]** Highest degree when using spherical harmonics.

enforce(*net*, **coordinates*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, *r*)

Re-parameterizes outputs such that the Dirichlet condition is satisfied on both spherical boundaries.

- If both inner and outer boundaries are specified $\mathbf{R}(r_0,\theta,\phi) = \mathbf{R}_0$ and $\mathbf{R}(r_1,\theta,\phi) = \mathbf{R}_1$.

  The re-parameterization is $\left(1-\tilde{r}\right)\mathbf{R}_0 + \tilde{r}\mathbf{R}_1 + \left(1 - e^{\tilde{r}(1-\tilde{r})}\right)\mathrm{ANN}(r)$ where $\tilde{r} = \dfrac{r - r_0}{r_1 - r_0}$;

- If only one boundary is specified (inner or outer) $\mathbf{R}(r_0,\theta,\phi) = \mathbf{R_0}$

  The re-parameterization is $\mathbf{R}_0 + \left(1 - e^{-|r-r_0|}\right)\mathrm{ANN}(r)$;

where ANN is the neural network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **r** (*torch.Tensor*) – The radii (or r-component) of the inputs to the network.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.conditions.**DoubleEndedBVP1D**(*x_min*, *x_max*, *x_min_val=None*, *x_min_prime=None*, *x_max_val=None*, *x_max_prime=None*)

Bases: *neurodiffeq.conditions.BaseCondition*

A boundary condition on a 1-D range where $x \in [x_0, x_1]$. The conditions should have the following parts:

- $u(x_0) = u_0$ or $u'_x(x_0) = u'_0$,
- $u(x_1) = u_1$ or $u'_x(x_1) = u'_1$,

where $u'_x = \dfrac{\partial u}{\partial x}$.

Parameters

- **x_min** (*float*) – The lower bound of x, the x_0.
- **x_max** (*float*) – The upper bound of x, the x_1.
- **x_min_val** (*callable, optional*) – The Dirichlet boundary condition when $x = x_0$, the $u(x_0)$, defaults to None.
- **x_min_prime** (*callable, optional*) – The Neumann boundary condition when $x = x_0$, the $u'_x(x_0)$, defaults to None.
- **x_max_val** (*callable, optional*) – The Dirichlet boundary condition when $x = x_1$, the $u(x_1)$, defaults to None.
- **x_max_prime** (*callable, optional*) – The Neumann boundary condition when $x = x_1$, the $u'_x(x_1)$, defaults to None.

Raises **NotImplementedError** – When unimplemented boundary conditions are configured.

Note: This condition cannot be passed to `neurodiffeq.conditions.EnsembleCondition` unless both boundaries uses Dirichlet conditions (by specifying only `x_min_val` and `x_max_val`) and `force` is set to True in EnsembleCondition's constructor.

---

enforce(*net*, *x*)

Enforces this condition on a network with inputs *x*.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **x** (*torch.Tensor*) – The x-coordinates of the samples; i.e., the spatial coordinates.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

Note: This method overrides the default method of `neurodiffeq.conditions.BaseCondition`. In general, you should avoid overriding `enforce` when implementing custom boundary conditions.

parameterize(*u*, *x*, **additional_tensors*)

Re-parameterizes outputs such that the boundary conditions are satisfied.

There are four boundary conditions that are currently implemented:

- For Dirichlet-Dirichlet boundary condition $u(x_0) = u_0$ and $u(x_1) = u_1$:

  The re-parameterization is $u(x) = A + \tilde{x}\big(1-\tilde{x}\big)\mathrm{ANN}(x)$, where $A = \big(1-\tilde{x}\big)u_0 + \big(\tilde{x}\big)u_1$.

- For Dirichlet-Neumann boundary condition $u(x_0) = u_0$ and $u'_x(x_1) = u'_1$:

  The re-parameterization is $u(x) = A(x) + \tilde{x}\Big(\mathrm{ANN}(x) - \mathrm{ANN}(x_1) + x_0 - \big(x_1 - x_0\big)\mathrm{ANN}'_x(x_1)\Big)$, where $A(x) = \big(1-\tilde{x}\big)u_0 + \dfrac{1}{2}\tilde{x}^2\big(x_1 - x_0\big)u'_1$.

- For Neumann-Dirichlet boundary condition $u'_x(x_0) = u'_0$ and $u(x_1) = u_1$:

  The re-parameterization is $u(x) = A(x) + \big(1-\tilde{x}\big)\Big(\mathrm{ANN}(x) - \mathrm{ANN}(x_0) + x_1 + \big(x_1 - x_0\big)\mathrm{ANN}'_x(x_0)\Big)$, where $A(x) = \tilde{x}u_1 - \dfrac{1}{2}\big(1-\tilde{x}\big)^2\big(x_1 - x_0\big)u'_0$.

- For Neumann-Neumann boundary condition $u'_x(x_0) = u'_0$ and $u'_x(x_1) = u'_1$:

  The re-parameterization is $u(x) = A(x) + \dfrac{1}{2}\tilde{x}^2\big(\mathrm{ANN}(x) - \mathrm{ANN}(x_1) - \dfrac{1}{2}\mathrm{ANN}'_x(x_1)\big(x_1 - x_0\big)\big), + \dfrac{1}{2}\big(1-\tilde{x}\big)^2\big(\mathrm{ANN}(x) - \mathrm{ANN}(x_0) + \dfrac{1}{2}\mathrm{ANN}'_x(x_0)\big(x_1 - x_0\big)\big)$, where $A(x) = \dfrac{1}{2}\tilde{x}^2\big(x_1 - x_0\big)u'_1 - \dfrac{1}{2}\big(1-\tilde{x}\big)^2\big(x_1 - x_0\big)u'_0$.

Notations:

- $\tilde{x} = \dfrac{x - x_0}{x_1 - x_0}$,
- ANN is the neural network,
- and $\mathrm{ANN}'_x = \dfrac{\partial ANN}{\partial x}$.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **x** (*torch.Tensor*) – The x-coordinates of the samples; i.e., the spatial coordinates.
- **additional_tensors** (*torch.Tensor*) – additional tensors that will be passed by `enforce`

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

class neurodiffeq.conditions.**EnsembleCondition**(**sub_conditions*, *force=False*)

Bases: *neurodiffeq.conditions.BaseCondition*

An ensemble condition that enforces sub-conditions on individual output units of the networks.

Parameters

- **sub_conditions** (BaseCondition) – Condition(s) to be ensemble'd.
- **force** (*bool*) – Whether or not to force ensembl'ing even when *.enforce* is overridden in one of the sub-conditions.

enforce(*net*, **coordinates*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, **input_tensors*)

Re-parameterizes each column in output_tensor individually, using its corresponding sub-condition. This is useful when solving differential equations with a single, multi-output network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network. Number of units (.shape[1]) must equal number of sub-conditions.
- **input_tensors** (*torch.Tensor*) – Inputs to the neural network; i.e., sampled coordinates; i.e., independent variables.

Returns The column-wise re-parameterized network output, concatenated across columns so that it's still one tensor.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

class neurodiffeq.conditions.**IBVP1D**(*x_min*, *x_max*, *t_min*, *t_min_val*, *x_min_val=None*, *x_min_prime=None*, *x_max_val=None*, *x_max_prime=None*)

Bases: *neurodiffeq.conditions.BaseCondition*

An initial & boundary condition on a 1-D range where $x \in [x_0, x_1]$ and time starts at t_0. The conditions should have the following parts:

- $u(x, t_0) = u_0(x)$,
- $u(x_0, t) = g(t)$ or $u'_x(x_0, t) = p(t)$,
- $u(x_1, t) = h(t)$ or $u'_x(x_1, t) = q(t)$,

where $u'_x = \dfrac{\partial u}{\partial x}$.

Parameters

- **x_min** (*float*) – The lower bound of x, the x_0.
- **x_max** (*float*) – The upper bound of x, the x_1.
- **t_min** (*float*) – The initial time, the t_0.
- **t_min_val** (*callable*) – The initial condition, the $u_0(x)$.
- **x_min_val** (*callable, optional*) – The Dirichlet boundary condition when $x = x_0$, the $u(x_0, t)$, defaults to None.
- **x_min_prime** (*callable, optional*) – The Neumann boundary condition when $x = x_0$, the $u'_x(x_0, t)$, defaults to None.
- **x_max_val** (*callable, optional*) – The Dirichlet boundary condition when $x = x_1$, the $u(x_1, t)$, defaults to None.
- **x_max_prime** (*callable, optional*) – The Neumann boundary condition when $x = x_1$, the $u'_x(x_1, t)$, defaults to None.

Raises **NotImplementedError** – When unimplemented boundary conditions are configured.

Note: This condition cannot be passed to `neurodiffeq.conditions.EnsembleCondition` unless both boundaries uses Dirichlet conditions (by specifying only `x_min_val` and `x_max_val`) and `force` is set to True in EnsembleCondition's constructor.

---

enforce(*net*, *x*, *t*)

Enforces this condition on a network with inputs *x* and *t*

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **x** (*torch.Tensor*) – The x-coordinates of the samples; i.e., the spatial coordinates.
- **t** (*torch.Tensor*) – The t-coordinates of the samples; i.e., the temporal coordinates.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

Note: This method overrides the default method of `neurodiffeq.conditions.BaseCondition`. In general, you should avoid overriding `enforce` when implementing custom boundary conditions.

---

parameterize(*u*, *x*, *t*, **additional_tensors*)

Re-parameterizes outputs such that the initial and boundary conditions are satisfied.

The Initial condition is always $u(x, t_0) = u_0(x)$. There are four boundary conditions that are currently implemented:

- For Dirichlet-Dirichlet boundary condition $u(x_0,t)=g(t)$ and $u(x_1,t)=h(t)$:

  The re-parameterization is $u(x,t) = A(x,t) + \tilde{x}\big(1-\tilde{x}\big)\Big(1-e^{-\tilde{t}}\Big)\mathrm{ANN}(x,t)$, where $A(x,t) = u_0(x) + \tilde{x}\big(h(t)-h(t_0)\big) + \big(1-\tilde{x}\big)\big(g(t)-g(t_0)\big)$.

- For Dirichlet-Neumann boundary condition $u(x_0,t)=g(t)$ and $u'_x(x_1,t)=q(t)$:

  The re-parameterization is $u(x,t) = A(x,t) + \tilde{x}\Big(1-e^{-\tilde{t}}\Big)\Big(\mathrm{ANN}(x,t) - \big(x_1-x_0\big)\mathrm{ANN}'_x(x_1,t) - \mathrm{ANN}(x_1,t)\Big)$, where $A(x,t) = u_0(x) + \big(x-x_0\big)\big(q(t)-q(t_0)\big) + \big(g(t)-g(t_0)\big)$.

- For Neumann-Dirichlet boundary condition $u'_x(x_0,t)=p(t)$ and $u(x_1,t)=h(t)$:

  The re-parameterization is $u(x,t) = A(x,t) + \big(1-\tilde{x}\big)\Big(1-e^{-\tilde{t}}\Big)\Big(\mathrm{ANN}(x,t) - \big(x_1-x_0\big)\mathrm{ANN}'_x(x_0,t) - \mathrm{ANN}(x_0,t)\Big)$, where $A(x,t) = u_0(x) + \big(x_1-x\big)\big(p(t)-p(t_0)\big) + \big(h(t)-h(t_0)\big)$.

- For Neumann-Neumann boundary condition $u'_x(x_0,t)=p(t)$ and $u'_x(x_1,t)=q(t)$

  The re-parameterization is $u(x,t) = A(x,t) + \Big(1-e^{-\tilde{t}}\Big)\Big(\mathrm{ANN}(x,t) - \big(x-x_0\big)\mathrm{ANN}'_x(x_0,t) + \frac{1}{2}\tilde{x}^2\big(x_1-x_0\big)\big(\mathrm{ANN}'_x(x_0,t) - \mathrm{ANN}'_x(x_1,t)\big)\Big)$, where $A(x,t) = u_0(x) - \frac{1}{2}\big(1-\tilde{x}\big)^2\big(x_1-x_0\big)\big(p(t)-p(t_0)\big) + \frac{1}{2}\tilde{x}^2\big(x_1-x_0\big)\big(q(t)-q(t_0)\big)$.

Notations:

- $\tilde{t} = \dfrac{t-t_0}{t_1-t_0}$,
- $\tilde{x} = \dfrac{x-x_0}{x_1-x_0}$,
- ANN is the neural network,
- and $\mathrm{ANN}'_x = \dfrac{\partial ANN}{\partial x}$.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **x** (*torch.Tensor*) – The x-coordinates of the samples; i.e., the spatial coordinates.
- **t** (*torch.Tensor*) – The t-coordinates of the samples; i.e., the temporal coordinates.
- **additional_tensors** (*torch.Tensor*) – additional tensors that will be passed by `enforce`

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.conditions.**IVP**(*t_0*, *u_0=None*, *u_0_prime=None*)

Bases: *neurodiffeq.conditions.BaseCondition*

An initial value problem of one of the following forms:

- Dirichlet condition: $u(t_0) = u_0$.
- Neumann condition: $\left.\frac{\partial u}{\partial t}\right|_{t=t_0} = u_0'$.

Parameters

- **t_0** (*float*) – The initial time.
- **u_0** (*float*) – The initial value of u. $u(t_0) = u_0$.
- **u_0_prime** (*float, optional*) – The initial derivative of u w.r.t. t. $\left.\frac{\partial u}{\partial t}\right|_{t=t_0} = u_0'$. Defaults to None.

enforce(*net*, **coordinates*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, *t*)

Re-parameterizes outputs such that the Dirichlet/Neumann condition is satisfied.

- For Dirichlet condition, the re-parameterization is $u(t) = u_0 + \left(1 - e^{-(t-t_0)}\right) \text{ANN}(t)$ where ANN is the neural network.
- For Neumann condition, the re-parameterization is $u(t) = u_0 + (t-t_0)u_0' + \left(1 - e^{-(t-t_0)}\right)^2 \text{ANN}(t)$ where ANN is the neural network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **t** (*torch.Tensor*) – Input to the neural network; i.e., sampled time-points; i.e., independent variables.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

class neurodiffeq.conditions.**InfDirichletBVPSpherical**(*r_0*, *f*, *g*, *order=1*)

Bases: *neurodiffeq.conditions.BaseCondition*

Similar to `neurodiffeq.conditions.DirichletBVPSpherical`. but with $r_1 \to +\infty$. Specifically,

- $u(r_0, \theta, \phi) = f(\theta, \phi)$,
- $\lim_{r \to +\infty} u(r, \theta, \phi) = g(\theta, \phi)$.

Parameters

- **r_0** (*float*) – The radius of the interior boundary. When $r_0 = 0$, the interior boundary collapses to a single point (center of the ball).
- **f** (*callable*) – The value of u on the interior boundary. $u(r_0, \theta, \phi) = f(\theta, \phi)$.
- **g** (*callable*) – The value of u at infinity. $\lim_{r \to +\infty} u(r, \theta, \phi) = g(\theta, \phi)$.
- **order** (*int or float*) – The smallest k such that $\lim_{r \to +\infty} u(r, \theta, \phi) e^{-kr} = 0$. Defaults to 1.

enforce(*net*, **coordinates*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, *r*, *theta*, *phi*)

Re-parameterizes outputs such that the Dirichlet condition is satisfied both at r_0 and infinity. The re-parameterization is

,

where ANN is the neural network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **r** (*torch.Tensor*) – The radii (or r-component) of the inputs to the network.
- **theta** (*torch.Tensor*) – The co-latitudes (or θ-component) of the inputs to the network.
- **phi** (*torch.Tensor*) – The longitudes (or ϕ-component) of the inputs to the network.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)

[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.conditions.**InfDirichletBVPSphericalBasis**(*r_0*, *R_0*, *R_inf*, *order=1*, *max_degree=None*)

Bases: *neurodiffeq.conditions.BaseCondition*

Similar to `neurodiffeq.conditions.InfDirichletBVPSpherical`. The only difference is this condition is enforced on a neural net that only takes in r and returns the spherical harmonic coefficients R(r). We constrain the coefficients $R_k(r)$ in $u(r,\theta,\phi) = \sum_k R_k(r)Y_k(\theta,\phi)$, where $\left\{Y_k(\theta,\phi)\right\}_{k=1}^{K}$ can be **any spherical function basis**. A recommended choice is the real spherical harmonics $Y_l^m(\theta,\phi)$, where l is the degree of the spherical harmonics and m is the order of the spherical harmonics.

The boundary conditions are: $\mathbf{R}(r_0) = \mathbf{R}_0$ and $\lim_{r_0\to+\infty}\mathbf{R}(r) = \mathbf{R}_1$, where $\mathbf{R}$ is a vector whose components are $\left\{R_k\right\}_{k=1}^{K}$.

Parameters

- **r_0** (*float*) – The radius of the interior boundary. When r_0 = 0, the interior boundary is collapsed to a single point (center of the ball).
- **R_0** (*torch.Tensor*) – The value of harmonic coefficients R on the interior boundary. $R(r_0) = R_0$.
- **R_inf** (*torch.Tensor*) – The value of harmonic coefficients R at infinity. $\lim_{r\to+\infty} R(r) = R_\infty$.
- **order** (*int or float*) – The smallest k that guarantees $\lim_{r\to+\infty} R(r)e^{-kr} = \mathbf{0}$. Defaults to 1.
- **max_degree** (*int*) – **[DEPRECATED]** Highest degree when using spherical harmonics.

enforce(*net*, **coordinates*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, *r*)

Re-parameterizes outputs such that the Dirichlet condition is satisfied at both r_0 and infinity.

The re-parameterization is

,

where ANN is the neural network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **r** (*torch.Tensor*) – The radii (or r-component) of the inputs to the network.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)
[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.conditions.**IrregularBoundaryCondition**
Bases: *neurodiffeq.conditions.BaseCondition*

enforce(*net*, **coordinates*)
Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

in_domain(**coordinates*)
Given the coordinates (numpy.ndarray), the methods returns an boolean array indicating whether the points lie within the domain.

Parameters **coordinates** (*numpy.ndarray*) – Input tensors, each with shape (n_samples, 1).

Returns Whether each point lies within the domain.

Return type *numpy.ndarray*

Note:

- This method is meant to be used by monitors for irregular domain visualization.

---

parameterize(*output_tensor*, **input_tensors*)
Re-parameterizes output(s) of a network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **input_tensors** (*torch.Tensor*) – Inputs to the neural network; i.e., sampled coordinates; i.e., independent variables.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

Note: This method is **abstract** for BaseCondition

---

set_impose_on(*ith_unit*)
[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.conditions.**NoCondition**
Bases: *neurodiffeq.conditions.BaseCondition*

A polymorphic condition where no re-parameterization will be performed.

Note: This condition is called *polymorphic* because it can be enforced on networks of arbitrary input/output sizes.

---

enforce(*net*, **coordinates*)
Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

parameterize(*output_tensor*, **input_tensors*)
Performs no re-parameterization, or identity parameterization, in this case.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **input_tensors** (*torch.Tensor*) – Inputs to the neural network; i.e., sampled coordinates; i.e., independent variables.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

set_impose_on(*ith_unit*)
[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

4.4 *neurodiffeq.solvers*

class neurodiffeq.solvers.**BaseSolution**(*nets*, *conditions*)
Bases: abc.ABC

A solution to a PDE/ODE (system).

Parameters

- **nets** (list[*torch.nn.Module*] or *torch.nn.Module*) – The neural networks that approximate the PDE/ODE solution.
  - If nets is a list of torch.nn.Module, it should have the same length with conditions
  - If nets is a single torch.nn.Module, it should have as many output units as length of conditions
- **conditions** (list[*neurodiffeq.conditions.BaseCondition*]) – A list of conditions that should be enforced on the PDE/ODE solution. conditions should have a length equal to the number of dependent variables in the ODE/PDE system.

class neurodiffeq.solvers.**BaseSolver**(*diff_eqs*, *conditions*, *nets=None*, *train_generator=None*, *valid_generator=None*, *analytic_solutions=None*, *optimizer=None*, *loss_fn=None*, *n_batches_train=1*, *n_batches_valid=4*, *metrics=None*, *n_input_units=None*, *n_output_units=None*, *shuffle=None*, *batch_size=None*)
Bases: abc.ABC, neurodiffeq.solvers_utils.PretrainedSolver

A class for solving ODE/PDE systems.

Parameters

- **diff_eqs** (*callable*) – The differential equation system to solve, which maps a tuple of coordinates to a tuple of ODE/PDE residuals. Both the coordinates and ODE/PDE residuals must have shape (-1, 1).
- **conditions** (list[*neurodiffeq.conditions.BaseCondition*]) – List of boundary conditions for each target function.
- **nets** (list[*torch.nn.Module*], optional) – List of neural networks for parameterized solution. If provided, length must equal that of conditions.
- **train_generator** (*neurodiffeq.generators.BaseGenerator*, required) – A generator for sampling training points. It must provide a *.get_examples()* method and a *.size* field.
- **valid_generator** (*neurodiffeq.generators.BaseGenerator*, required) – A generator for sampling validation points. It must provide a *.get_examples()* method and a *.size* field.
- **analytic_solutions** (*callable, optional*) – **[DEPRECATED]** Pass metrics instead. The analytical solutions to be compared with neural net solutions. It maps a tuple of coordinates to a tuple of function values. The output shape should match that of networks.
- **optimizer** (*torch.nn.optim.Optimizer*, optional) – The optimizer to be used for training.
- **loss_fn** (str or *torch.nn.moduesl.loss._Loss* or callable) – The loss function used for training.
  - If a str, must be present in the keys of *neurodiffeq.losses._losses*.
  - If a *torch.nn.modules.loss._Loss* instance, just pass the instance.

- If any other callable, it must map A) a residual tensor (shape *(n_points, n_equations)*), B) a function values tuple (length *n_funcs*, each element a tensor of shape *(n_points, 1)*), and C) a coordinate values tuple (length *n_coords*, each element a tensor of shape *(n_coords, 1)* to a tensor of empty shape (i.e. a scalar). The returned tensor must be connected to the computational graph, so that backpropagation can be performed.

- **n_batches_train** (*int, optional*) – Number of batches to train in every epoch, where batch-size equals `train_generator.size`. Defaults to 1.
- **n_batches_valid** (*int, optional*) – Number of batches to validate in every epoch, where batch-size equals `valid_generator.size`. Defaults to 4.
- **metrics** (*dict, optional*) – Additional metrics to be logged (besides loss). `metrics` should be a dict where
  - Keys are metric names (e.g. 'analytic_mse');
  - Values are functions (callables) that computes the metric value. These functions must accept the same input as the differential equation `diff_eq`.
- **n_input_units** (*int, required*) – Number of input units for each neural network. Ignored if `nets` is specified.
- **n_output_units** (*int, required*) – Number of output units for each neural network. Ignored if `nets` is specified.
- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify `n_batches_train` and `n_batches_valid` instead.
- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

additional_loss(*residual*, *funcs*, *coords*)

Additional loss terms for training. This method is to be overridden by subclasses. This method can use any of the internal variables: self.nets, self.conditions, self.global_epoch, etc.

Parameters

- **residual** (*torch.Tensor*) – Residual tensor of differential equation. It has shape (N_SAMPLES, N_EQUATIONS)
- **funcs** (*List[torch.Tensor]*) – Outputs of the networks after parameterization. There are `len(nets)` entries in total. Each entry is a tensor of shape (N_SAMPLES, N_OUTPUT_UNITS).
- **coords** (*List[torch.Tensor]*) – Inputs to the networks; a.k.a. the spatio-temporal coordinates of the system. There are `N_COORDS` entries in total. Each entry is a tensor of shape (N_SAMPLES, 1).

Returns Additional loss. Must be a `torch.Tensor` of empty shape (scalar).

Return type torch.Tensor

compute_func_val(*net*, *cond*, **coordinates*)

Compute the function value evaluated on the points specified by `coordinates`.

Parameters

- **net** (*torch.nn.Module*) – The network to be parameterized and evaluated.
- **cond** (*neurodiffeq.conditions.BaseCondition*) – The condition (a.k.a. parameterization) for the network.

- **coordinates** (*tuple[torch.Tensor]*) – A tuple of coordinate components, each with shape = (-1, 1).

Returns Function values at the sampled points.

Return type torch.Tensor

fit(*max_epochs*, *callbacks=()*, *tqdm_file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*, ***kwargs*)
Run multiple epochs of training and validation, update best loss at the end of each epoch.

If `callbacks` is passed, callbacks are run, one at a time, after training, validating and updating best model.

Parameters

- **max_epochs** (*int*) – Number of epochs to run.
- **callbacks** – A list of callback functions. Each function should accept the `solver` instance itself as its **only** argument.
- **tqdm_file** (*io.StringIO or _io.TextIOWrapper*) – File to write tqdm progress bar. If set to None, tqdm is not used at all. Defaults to `sys.stderr`.

Rtype callbacks list[callable]

Note:

1. This method does not return solution, which is done in the `.get_solution()` method.
2. A callback `cb(solver)` can set `solver._stop_training` to True to perform early stopping.

---

get_internals(*var_names=None*, *return_type='list'*)
Return internal variable(s) of the solver

- If var_names == 'all', return all internal variables as a dict.
- If var_names is single str, return the corresponding variables.
- If var_names is a list and return_type == 'list', return corresponding internal variables as a list.
- If var_names is a list and return_type == 'dict', return a dict with keys in var_names.

Parameters

- **var_names** (*str or list[str]*) – An internal variable name or a list of internal variable names.
- **return_type** (*str*) – {'list', 'dict'}; Ignored if `var_names` is a string.

Returns A single variable, or a list/dict of internal variables as indicated above.

Return type list or dict or any

get_residuals(**coords*, *to_numpy=False*, *best=True*, *no_reshape=False*)
Get the residuals of the differential equation (or system of differential equations) evaluated at given points.

Parameters

- **coords** (*tuple[torch.Tensor] or tuple[np.ndarray]*) – The coordinate values where the residual(s) shall be evaluated. If numpy arrays are passed, the method implicitly creates torch tensors with corresponding values.
- **to_numpy** (*bool*) – Whether to return numpy arrays. Defaults to False.

- **best** (*bool*) – If set to False, the network from the most recent epoch will be used to evaluate the residuals. If set to True, the network from the epoch with the lowest validation loss will be used to evaluate the residuals. Defaults to True.
- **no_reshape** (*bool*) – If set to True, no reshaping will be performed on output. Defaults to False.

Returns The residuals evaluated at given points. If there is only one equation in the differential equation system, a single torch tensor (or numpy array) will be returned. If there are multiple equations, a list of torch tensors (or numpy arrays) will be returned. The returned shape will be the same as the first input coordinate, unless *no_reshape* is set to True. Note that the return value will always be torch tensors (even if `coords` are numpy arrays) unless *to_numpy* is explicitly set to True.

Return type list[*torch.Tensor* or *numpy.array*] or *torch.Tensor* or *numpy.array*

get_solution(*copy=True*, *best=True*)

Get a (callable) solution object. See this usage example:

```
solution = solver.get_solution()
point_coords = train_generator.get_examples()
value_at_points = solution(point_coords)
```

Parameters

- **copy** (*bool*) – Whether to make a copy of the networks so that subsequent training doesn't affect the solution; Defaults to True.
- **best** (*bool*) – Whether to return the solution with lowest loss instead of the solution after the last epoch. Defaults to True.

Returns A solution object which can be called. To evaluate the solution on certain points, you should pass the coordinates vector(s) to the returned solution.

Return type *BaseSolution*

global_epoch

Global epoch count, always equal to the length of train loss history.

Returns Number of training epochs that have been run.

Return type int

run_train_epoch()

Run a training epoch, update history, and perform gradient descent.

run_valid_epoch()

Run a validation epoch and update history.

class neurodiffeq.solvers.**BundleSolution1D**(*nets*, *conditions*)

Bases: *neurodiffeq.solvers.BaseSolution*

class neurodiffeq.solvers.**BundleSolver1D**(*ode_system*, *conditions*, *t_min*, *t_max*, *theta_min=None*, *theta_max=None*, *nets=None*, *train_generator=None*, *valid_generator=None*, *analytic_solutions=None*, *optimizer=None*, *loss_fn=None*, *n_batches_train=1*, *n_batches_valid=4*, *metrics=None*, *n_output_units=1*, *batch_size=None*, *shuffle=None*)

Bases: *neurodiffeq.solvers.BaseSolver*

A solver class for solving ODEs (single-input differential equations) , or a bundle of ODEs for different values of its parameters and/or conditions

Parameters

- **ode_system** (*callable*) – The ODE system to solve, which maps a torch.Tensor or a tuple of torch.Tensors, to a tuple of ODE residuals, both the input and output must have shape (n_samples, 1).
- **conditions** (list[*neurodiffeq.conditions.BaseCondition*]) – List of conditions for each target function.
- **t_min** (*float, optional*) – Lower bound of input (start time). Ignored if `train_generator` and `valid_generator` are both set.
- **t_max** (*float, optional*) – Upper bound of input (start time). Ignored if `train_generator` and `valid_generator` are both set.
- **theta_min** (*float or tuple, optional*) – Lower bound of input (parameters and/or conditions). If conditions are included in the bundle, the order should match the one inferred by the values of the `bundle_conditions` input in the `neurodiffeq.conditions.BundleIVP`. Defaults to None. Ignored if `train_generator` and `valid_generator` are both set.
- **theta_max** (*float or tuple, optional*) – Upper bound of input (parameters and/or conditions). If conditions are included in the bundle, the order should match the one inferred by the values of the `bundle_conditions` input in the `neurodiffeq.conditions.BundleIVP`. Defaults to None. Ignored if `train_generator` and `valid_generator` are both set.
- **nets** (*list[torch.nn.Module], optional*) – List of neural networks for parameterized solution. If provided, length of `nets` must equal that of `conditions`
- **train_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – Generator for sampling training points, which must provide a `.get_examples()` method and a `.size` field. `train_generator` must be specified if `t_min` and `t_max` are not set.
- **valid_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – Generator for sampling validation points, which must provide a `.get_examples()` method and a `.size` field. `valid_generator` must be specified if `t_min` and `t_max` are not set.
- **analytic_solutions** (*callable, optional*) – Analytical solutions to be compared with neural net solutions. It maps a torch.Tensor to a tuple of function values. Output shape should match that of `nets`.
- **optimizer** (`torch.nn.optim.Optimizer`, optional) – Optimizer to be used for training. Defaults to a `torch.optim.Adam` instance that trains on all parameters of `nets`.
- **loss_fn** (str or *torch.nn.moduesl.loss._Loss* or callable) – The loss function used for training.
  - If a str, must be present in the keys of *neurodiffeq.losses._losses*.
  - If a *torch.nn.modules.loss._Loss* instance, just pass the instance.
  - If any other callable, it must map A) a residual tensor (shape *(n_points, n_equations)*), B) a function values tuple (length *n_funcs*, each element a tensor of shape *(n_points, 1)*), and C) a coordinate values tuple (length *n_coords*, each element a tensor of shape *(n_coords, 1)* to a tensor of empty shape (i.e. a scalar). The returned tensor must be connected to the computational graph, so that backpropagation can be performed.

- **n_batches_train** (*int, optional*) – Number of batches to train in every epoch, where batch-size equals `train_generator.size`. Defaults to 1.
- **n_batches_valid** (*int, optional*) – Number of batches to validate in every epoch, where batch-size equals `valid_generator.size`. Defaults to 4.
- **metrics** (*dict[str, callable], optional*) – Additional metrics to be logged (besides loss). `metrics` should be a dict where
  - Keys are metric names (e.g. 'analytic_mse');
  - Values are functions (callables) that computes the metric value. These functions must accept the same input as the differential equation `ode_system`.
- **n_output_units** (*int, optional*) – Number of output units for each neural network. Ignored if `nets` is specified. Defaults to 1.
- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify `n_batches_train` and `n_batches_valid` instead.
- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

additional_loss(*residual*, *funcs*, *coords*)

Additional loss terms for training. This method is to be overridden by subclasses. This method can use any of the internal variables: self.nets, self.conditions, self.global_epoch, etc.

Parameters

- **residual** (*torch.Tensor*) – Residual tensor of differential equation. It has shape (N_SAMPLES, N_EQUATIONS)
- **funcs** (*List[torch.Tensor]*) – Outputs of the networks after parameterization. There are `len(nets)` entries in total. Each entry is a tensor of shape (N_SAMPLES, N_OUTPUT_UNITS).
- **coords** (*List[torch.Tensor]*) – Inputs to the networks; a.k.a. the spatio-temporal coordinates of the system. There are `N_COORDS` entries in total. Each entry is a tensor of shape (N_SAMPLES, 1).

Returns Additional loss. Must be a `torch.Tensor` of empty shape (scalar).

Return type torch.Tensor

compute_func_val(*net*, *cond*, **coordinates*)

Compute the function value evaluated on the points specified by `coordinates`.

Parameters

- **net** (*torch.nn.Module*) – The network to be parameterized and evaluated.
- **cond** (*neurodiffeq.conditions.BaseCondition*) – The condition (a.k.a. parameterization) for the network.
- **coordinates** (*tuple[torch.Tensor]*) – A tuple of coordinate components, each with shape = (-1, 1).

Returns Function values at the sampled points.

Return type torch.Tensor

fit(*max_epochs*, *callbacks=()*, *tqdm_file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*, ***kwargs*)

Run multiple epochs of training and validation, update best loss at the end of each epoch.

If `callbacks` is passed, callbacks are run, one at a time, after training, validating and updating best model.

Parameters

- **max_epochs** (*int*) – Number of epochs to run.
- **callbacks** – A list of callback functions. Each function should accept the `solver` instance itself as its **only** argument.
- **tqdm_file** (*io.StringIO or _io.TextIOWrapper*) – File to write tqdm progress bar. If set to None, tqdm is not used at all. Defaults to `sys.stderr`.

Rtype callbacks list[callable]

Note:

1. This method does not return solution, which is done in the `.get_solution()` method.
2. A callback `cb(solver)` can set `solver._stop_training` to True to perform early stopping.

---

get_internals(*var_names=None*, *return_type='list'*)
Return internal variable(s) of the solver

- If var_names == 'all', return all internal variables as a dict.
- If var_names is single str, return the corresponding variables.
- If var_names is a list and return_type == 'list', return corresponding internal variables as a list.
- If var_names is a list and return_type == 'dict', return a dict with keys in var_names.

Parameters

- **var_names** (*str or list[str]*) – An internal variable name or a list of internal variable names.
- **return_type** (*str*) – {'list', 'dict'}; Ignored if `var_names` is a string.

Returns A single variable, or a list/dict of internal variables as indicated above.

Return type list or dict or any

get_residuals(**coords*, *to_numpy=False*, *best=True*, *no_reshape=False*)
Get the residuals of the differential equation (or system of differential equations) evaluated at given points.

Parameters

- **coords** (*tuple[torch.Tensor] or tuple[np.ndarray]*) – The coordinate values where the residual(s) shall be evaluated. If numpy arrays are passed, the method implicitly creates torch tensors with corresponding values.
- **to_numpy** (*bool*) – Whether to return numpy arrays. Defaults to False.
- **best** (*bool*) – If set to False, the network from the most recent epoch will be used to evaluate the residuals. If set to True, the network from the epoch with the lowest validation loss will be used to evaluate the residuals. Defaults to True.
- **no_reshape** (*bool*) – If set to True, no reshaping will be performed on output. Defaults to False.

Returns The residuals evaluated at given points. If there is only one equation in the differential equation system, a single torch tensor (or numpy array) will be returned. If there are multiple equations, a list of torch tensors (or numpy arrays) will be returned. The returned shape will be the same as the first input coordinate, unless *no_reshape* is set to True. Note that the return value will always be torch tensors (even if `coords` are numpy arrays) unless *to_numpy* is explicitly set to True.

Return type list[*torch.Tensor* or *numpy.array*] or *torch.Tensor* or *numpy.array*

get_solution(*copy=True*, *best=True*)
Get a (callable) solution object. See this usage example:

```
solution = solver.get_solution()
point_coords = train_generator.get_examples()
value_at_points = solution(point_coords)
```

Parameters

- **copy** (*bool*) – Whether to make a copy of the networks so that subsequent training doesn't affect the solution; Defaults to True.
- **best** (*bool*) – Whether to return the solution with lowest loss instead of the solution after the last epoch. Defaults to True.

Returns A solution object which can be called. To evaluate the solution on certain points, you should pass the coordinates vector(s) to the returned solution.

Return type *BaseSolution*

global_epoch
Global epoch count, always equal to the length of train loss history.

Returns Number of training epochs that have been run.

Return type int

run_train_epoch()
Run a training epoch, update history, and perform gradient descent.

run_valid_epoch()
Run a validation epoch and update history.

class neurodiffeq.solvers.**GenericSolution**(*nets*, *conditions*)
Bases: *neurodiffeq.solvers.BaseSolution*

class neurodiffeq.solvers.**GenericSolver**(*diff_eqs*, *conditions*, *nets=None*, *train_generator=None*, *valid_generator=None*, *analytic_solutions=None*, *optimizer=None*, *loss_fn=None*, *n_batches_train=1*, *n_batches_valid=4*, *metrics=None*, *n_input_units=None*, *n_output_units=None*, *shuffle=None*, *batch_size=None*)
Bases: *neurodiffeq.solvers.BaseSolver*

additional_loss(*residual*, *funcs*, *coords*)
Additional loss terms for training. This method is to be overridden by subclasses. This method can use any of the internal variables: self.nets, self.conditions, self.global_epoch, etc.

Parameters

- **residual** (*torch.Tensor*) – Residual tensor of differential equation. It has shape (N_SAMPLES, N_EQUATIONS)

- **funcs** (*List[torch.Tensor]*) – Outputs of the networks after parameterization. There are `len(nets)` entries in total. Each entry is a tensor of shape (N_SAMPLES, N_OUTPUT_UNITS).
- **coords** (*List[torch.Tensor]*) – Inputs to the networks; a.k.a. the spatio-temporal coordinates of the system. There are `N_COORDS` entries in total. Each entry is a tensor of shape (N_SAMPLES, 1).

Returns Additional loss. Must be a `torch.Tensor` of empty shape (scalar).

Return type torch.Tensor

compute_func_val(*net*, *cond*, **coordinates*)
Compute the function value evaluated on the points specified by `coordinates`.

Parameters

- **net** (*torch.nn.Module*) – The network to be parameterized and evaluated.
- **cond** (*neurodiffeq.conditions.BaseCondition*) – The condition (a.k.a. parameterization) for the network.
- **coordinates** (*tuple[torch.Tensor]*) – A tuple of coordinate components, each with shape = (-1, 1).

Returns Function values at the sampled points.

Return type torch.Tensor

fit(*max_epochs*, *callbacks=()*, *tqdm_file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*, ***kwargs*)
Run multiple epochs of training and validation, update best loss at the end of each epoch.

If `callbacks` is passed, callbacks are run, one at a time, after training, validating and updating best model.

Parameters

- **max_epochs** (*int*) – Number of epochs to run.
- **callbacks** – A list of callback functions. Each function should accept the `solver` instance itself as its **only** argument.
- **tqdm_file** (*io.StringIO or _io.TextIOWrapper*) – File to write tqdm progress bar. If set to None, tqdm is not used at all. Defaults to `sys.stderr`.

Rtype callbacks list[callable]

Note:

1. This method does not return solution, which is done in the `.get_solution()` method.
2. A callback `cb(solver)` can set `solver._stop_training` to True to perform early stopping.

---

get_internals(*var_names=None*, *return_type='list'*)
Return internal variable(s) of the solver

- If var_names == 'all', return all internal variables as a dict.
- If var_names is single str, return the corresponding variables.
- If var_names is a list and return_type == 'list', return corresponding internal variables as a list.
- If var_names is a list and return_type == 'dict', return a dict with keys in var_names.

Parameters

- **var_names** (*str or list[str]*) – An internal variable name or a list of internal variable names.
- **return_type** (*str*) – {'list', 'dict'}; Ignored if `var_names` is a string.

Returns A single variable, or a list/dict of internal variables as indicated above.

Return type list or dict or any

get_residuals(**coords*, *to_numpy=False*, *best=True*, *no_reshape=False*)

Get the residuals of the differential equation (or system of differential equations) evaluated at given points.

Parameters

- **coords** (*tuple[torch.Tensor] or tuple[np.ndarray]*) – The coordinate values where the residual(s) shall be evaluated. If numpy arrays are passed, the method implicitly creates torch tensors with corresponding values.
- **to_numpy** (*bool*) – Whether to return numpy arrays. Defaults to False.
- **best** (*bool*) – If set to False, the network from the most recent epoch will be used to evaluate the residuals. If set to True, the network from the epoch with the lowest validation loss will be used to evaluate the residuals. Defaults to True.
- **no_reshape** (*bool*) – If set to True, no reshaping will be performed on output. Defaults to False.

Returns The residuals evaluated at given points. If there is only one equation in the differential equation system, a single torch tensor (or numpy array) will be returned. If there are multiple equations, a list of torch tensors (or numpy arrays) will be returned. The returned shape will be the same as the first input coordinate, unless *no_reshape* is set to True. Note that the return value will always be torch tensors (even if `coords` are numpy arrays) unless *to_numpy* is explicitly set to True.

Return type list[*torch.Tensor* or *numpy.array*] or *torch.Tensor* or *numpy.array*

get_solution(*copy=True*, *best=True*)

Get a (callable) solution object. See this usage example:

```
solution = solver.get_solution()
point_coords = train_generator.get_examples()
value_at_points = solution(point_coords)
```

Parameters

- **copy** (*bool*) – Whether to make a copy of the networks so that subsequent training doesn't affect the solution; Defaults to True.
- **best** (*bool*) – Whether to return the solution with lowest loss instead of the solution after the last epoch. Defaults to True.

Returns A solution object which can be called. To evaluate the solution on certain points, you should pass the coordinates vector(s) to the returned solution.

Return type *BaseSolution*

global_epoch

Global epoch count, always equal to the length of train loss history.

Returns Number of training epochs that have been run.

Return type int

run_train_epoch()
Run a training epoch, update history, and perform gradient descent.

run_valid_epoch()
Run a validation epoch and update history.

class neurodiffeq.solvers.**Solution1D**(*nets*, *conditions*)
Bases: *neurodiffeq.solvers.BaseSolution*

class neurodiffeq.solvers.**Solution2D**(*nets*, *conditions*)
Bases: *neurodiffeq.solvers.BaseSolution*

class neurodiffeq.solvers.**SolutionSpherical**(*nets*, *conditions*)
Bases: *neurodiffeq.solvers.BaseSolution*

class neurodiffeq.solvers.**SolutionSphericalHarmonics**(*nets*, *conditions*, *max_degree=None*, *harmonics_fn=None*)
Bases: *neurodiffeq.solvers.SolutionSpherical*

A solution to a PDE (system) in spherical coordinates.

Parameters

- **nets** (list[*torch.nn.Module*]) – List of networks that takes in radius tensor and outputs the coefficients of spherical harmonics.
- **conditions** (list[*neurodiffeq.conditions.BaseCondition*]) – List of conditions to be enforced on each nets; must be of the same length as nets.
- **harmonics_fn** (*callable*) – Mapping from θ and ϕ to basis functions, e.g., spherical harmonics.
- **max_degree** (*int*) – **DEPRECATED and SUPERSEDED** by `harmonics_fn`. Highest used for the harmonic basis.

class neurodiffeq.solvers.**Solver1D**(*ode_system*, *conditions*, *t_min=None*, *t_max=None*, *nets=None*, *train_generator=None*, *valid_generator=None*, *analytic_solutions=None*, *optimizer=None*, *loss_fn=None*, *n_batches_train=1*, *n_batches_valid=4*, *metrics=None*, *n_output_units=1*, *batch_size=None*, *shuffle=None*)
Bases: *neurodiffeq.solvers.BaseSolver*

A solver class for solving ODEs (single-input differential equations)

Parameters

- **ode_system** (*callable*) – The ODE system to solve, which maps a torch.Tensor to a tuple of ODE residuals, both the input and output must have shape (n_samples, 1).
- **conditions** (list[*neurodiffeq.conditions.BaseCondition*]) – List of conditions for each target function.
- **t_min** (*float, optional*) – Lower bound of input (start time). Ignored if `train_generator` and `valid_generator` are both set.
- **t_max** (*float, optional*) – Upper bound of input (start time). Ignored if `train_generator` and `valid_generator` are both set.
- **nets** (*list[torch.nn.Module], optional*) – List of neural networks for parameterized solution. If provided, length of `nets` must equal that of `conditions`

- **train_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – Generator for sampling training points, which must provide a `.get_examples()` method and a `.size` field. `train_generator` must be specified if `t_min` and `t_max` are not set.
- **valid_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – Generator for sampling validation points, which must provide a `.get_examples()` method and a `.size` field. `valid_generator` must be specified if `t_min` and `t_max` are not set.
- **analytic_solutions** (*callable, optional*) – Analytical solutions to be compared with neural net solutions. It maps a torch.Tensor to a tuple of function values. Output shape should match that of `nets`.
- **optimizer** (`torch.nn.optim.Optimizer`, optional) – Optimizer to be used for training. Defaults to a `torch.optim.Adam` instance that trains on all parameters of `nets`.
- **loss_fn** (str or *torch.nn.moduesl.loss._Loss* or callable) – The loss function used for training.
  - If a str, must be present in the keys of *neurodiffeq.losses._losses*.
  - If a *torch.nn.modules.loss._Loss* instance, just pass the instance.
  - If any other callable, it must map A) a residual tensor (shape *(n_points, n_equations)*), B) a function values tuple (length *n_funcs*, each element a tensor of shape *(n_points, 1)*), and C) a coordinate values tuple (length *n_coords*, each element a tensor of shape *(n_coords, 1)* to a tensor of empty shape (i.e. a scalar). The returned tensor must be connected to the computational graph, so that backpropagation can be performed.
- **n_batches_train** (*int, optional*) – Number of batches to train in every epoch, where batch-size equals `train_generator.size`. Defaults to 1.
- **n_batches_valid** (*int, optional*) – Number of batches to validate in every epoch, where batch-size equals `valid_generator.size`. Defaults to 4.
- **metrics** (*dict[str, callable], optional*) – Additional metrics to be logged (besides loss). `metrics` should be a dict where
  - Keys are metric names (e.g. 'analytic_mse');
  - Values are functions (callables) that computes the metric value. These functions must accept the same input as the differential equation `ode_system`.
- **n_output_units** (*int, optional*) – Number of output units for each neural network. Ignored if `nets` is specified. Defaults to 1.
- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify `n_batches_train` and `n_batches_valid` instead.
- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

additional_loss(*residual*, *funcs*, *coords*)

Additional loss terms for training. This method is to be overridden by subclasses. This method can use any of the internal variables: self.nets, self.conditions, self.global_epoch, etc.

Parameters

- **residual** (*torch.Tensor*) – Residual tensor of differential equation. It has shape (N_SAMPLES, N_EQUATIONS)

- **funcs** (*List[torch.Tensor]*) – Outputs of the networks after parameterization. There are `len(nets)` entries in total. Each entry is a tensor of shape (N_SAMPLES, N_OUTPUT_UNITS).
- **coords** (*List[torch.Tensor]*) – Inputs to the networks; a.k.a. the spatio-temporal coordinates of the system. There are `N_COORDS` entries in total. Each entry is a tensor of shape (N_SAMPLES, 1).

Returns Additional loss. Must be a `torch.Tensor` of empty shape (scalar).

Return type torch.Tensor

compute_func_val(*net*, *cond*, **coordinates*)

Compute the function value evaluated on the points specified by `coordinates`.

Parameters

- **net** (*torch.nn.Module*) – The network to be parameterized and evaluated.
- **cond** (*neurodiffeq.conditions.BaseCondition*) – The condition (a.k.a. parameterization) for the network.
- **coordinates** (*tuple[torch.Tensor]*) – A tuple of coordinate components, each with shape = (-1, 1).

Returns Function values at the sampled points.

Return type torch.Tensor

fit(*max_epochs*, *callbacks=()*, *tqdm_file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*, ***kwargs*)

Run multiple epochs of training and validation, update best loss at the end of each epoch.

If `callbacks` is passed, callbacks are run, one at a time, after training, validating and updating best model.

Parameters

- **max_epochs** (*int*) – Number of epochs to run.
- **callbacks** – A list of callback functions. Each function should accept the `solver` instance itself as its **only** argument.
- **tqdm_file** (*io.StringIO or _io.TextIOWrapper*) – File to write tqdm progress bar. If set to None, tqdm is not used at all. Defaults to `sys.stderr`.

Rtype callbacks list[callable]

Note:

1. This method does not return solution, which is done in the `.get_solution()` method.
2. A callback `cb(solver)` can set `solver._stop_training` to True to perform early stopping.

---

get_internals(*var_names=None*, *return_type='list'*)

Return internal variable(s) of the solver

- If var_names == 'all', return all internal variables as a dict.
- If var_names is single str, return the corresponding variables.
- If var_names is a list and return_type == 'list', return corresponding internal variables as a list.
- If var_names is a list and return_type == 'dict', return a dict with keys in var_names.

Parameters

- **var_names** (*str or list[str]*) – An internal variable name or a list of internal variable names.
- **return_type** (*str*) – {'list', 'dict'}; Ignored if `var_names` is a string.

Returns A single variable, or a list/dict of internal variables as indicated above.

Return type list or dict or any

get_residuals(**coords*, *to_numpy=False*, *best=True*, *no_reshape=False*)

Get the residuals of the differential equation (or system of differential equations) evaluated at given points.

Parameters

- **coords** (*tuple[torch.Tensor] or tuple[np.ndarray]*) – The coordinate values where the residual(s) shall be evaluated. If numpy arrays are passed, the method implicitly creates torch tensors with corresponding values.
- **to_numpy** (*bool*) – Whether to return numpy arrays. Defaults to False.
- **best** (*bool*) – If set to False, the network from the most recent epoch will be used to evaluate the residuals. If set to True, the network from the epoch with the lowest validation loss will be used to evaluate the residuals. Defaults to True.
- **no_reshape** (*bool*) – If set to True, no reshaping will be performed on output. Defaults to False.

Returns The residuals evaluated at given points. If there is only one equation in the differential equation system, a single torch tensor (or numpy array) will be returned. If there are multiple equations, a list of torch tensors (or numpy arrays) will be returned. The returned shape will be the same as the first input coordinate, unless *no_reshape* is set to True. Note that the return value will always be torch tensors (even if `coords` are numpy arrays) unless *to_numpy* is explicitly set to True.

Return type list[*torch.Tensor* or *numpy.array*] or *torch.Tensor* or *numpy.array*

get_solution(*copy=True*, *best=True*)

Get a (callable) solution object. See this usage example:

```
solution = solver.get_solution()
point_coords = train_generator.get_examples()
value_at_points = solution(point_coords)
```

Parameters

- **copy** (*bool*) – Whether to make a copy of the networks so that subsequent training doesn't affect the solution; Defaults to True.
- **best** (*bool*) – Whether to return the solution with lowest loss instead of the solution after the last epoch. Defaults to True.

Returns A solution object which can be called. To evaluate the solution on certain points, you should pass the coordinates vector(s) to the returned solution.

Return type *BaseSolution*

global_epoch

Global epoch count, always equal to the length of train loss history.

Returns Number of training epochs that have been run.

Return type int

run_train_epoch()
Run a training epoch, update history, and perform gradient descent.

run_valid_epoch()
Run a validation epoch and update history.

class neurodiffeq.solvers.**Solver2D**(*pde_system*, *conditions*, *xy_min=None*, *xy_max=None*, *nets=None*, *train_generator=None*, *valid_generator=None*, *analytic_solutions=None*, *optimizer=None*, *loss_fn=None*, *n_batches_train=1*, *n_batches_valid=4*, *metrics=None*, *n_output_units=1*, *batch_size=None*, *shuffle=None*)

Bases: *neurodiffeq.solvers.BaseSolver*

A solver class for solving PDEs in 2 dimensions.

Parameters

- **pde_system** (*callable*) – The PDE system to solve, which maps two `torch.Tensor``s to PDE residuals (``tuple[torch.Tensor]`), both the input and output must have shape (n_samples, 1).
- **conditions** (list[*neurodiffeq.conditions.BaseCondition*]) – List of conditions for each target function.
- **xy_min** (*tuple[float, float], optional*) – The lower bound of 2 dimensions. If we only care about $x \geq x_0$ and $y \geq y_0$, then *xy_min* is *(x_0, y_0)*. Only needed when train_generator or valid_generator are not specified. Defaults to None
- **xy_max** (*tuple[float, float], optional*) – The upper bound of 2 dimensions. If we only care about $x \leq x_1$ and $y \leq y_1$, then *xy_min* is *(x_1, y_1)*. Only needed when train_generator or valid_generator are not specified. Defaults to None
- **nets** (*list[torch.nn.Module], optional*) – List of neural networks for parameterized solution. If provided, length of `nets` must equal that of `conditions`
- **train_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – Generator for sampling training points, which must provide a `.get_examples()` method and a `.size` field. `train_generator` must be specified if `t_min` and `t_max` are not set.
- **valid_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – Generator for sampling validation points, which must provide a `.get_examples()` method and a `.size` field. `valid_generator` must be specified if `t_min` and `t_max` are not set.
- **analytic_solutions** (*callable, optional*) – Analytical solutions to be compared with neural net solutions. It maps a torch.Tensor to a tuple of function values. Output shape should match that of `nets`.
- **optimizer** (`torch.nn.optim.Optimizer`, optional) – Optimizer to be used for training. Defaults to a `torch.optim.Adam` instance that trains on all parameters of `nets`.
- **loss_fn** (str or *torch.nn.moduesl.loss._Loss* or callable) – The loss function used for training.
  - If a str, must be present in the keys of *neurodiffeq.losses._losses*.
  - If a *torch.nn.modules.loss._Loss* instance, just pass the instance.
  - If any other callable, it must map A) a residual tensor (shape *(n_points, n_equations)*), B) a function values tuple (length *n_funcs*, each element a tensor of shape *(n_points, 1)*), and

C) a coordinate values tuple (length *n_coords*, each element a tensor of shape *(n_coords, 1)* to a tensor of empty shape (i.e. a scalar). The returned tensor must be connected to the computational graph, so that backpropagation can be performed.

- **n_batches_train** (*int, optional*) – Number of batches to train in every epoch, where batch-size equals `train_generator.size`. Defaults to 1.
- **n_batches_valid** (*int, optional*) – Number of batches to validate in every epoch, where batch-size equals `valid_generator.size`. Defaults to 4.
- **metrics** (*dict[str, callable], optional*) – Additional metrics to be logged (besides loss). `metrics` should be a dict where
  - Keys are metric names (e.g. 'analytic_mse');
  - Values are functions (callables) that computes the metric value. These functions must accept the same input as the differential equation `ode_system`.
- **n_output_units** (*int, optional*) – Number of output units for each neural network. Ignored if `nets` is specified. Defaults to 1.
- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify `n_batches_train` and `n_batches_valid` instead.
- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

additional_loss (*residual*, *funcs*, *coords*)

Additional loss terms for training. This method is to be overridden by subclasses. This method can use any of the internal variables: self.nets, self.conditions, self.global_epoch, etc.

Parameters

- **residual** (*torch.Tensor*) – Residual tensor of differential equation. It has shape (N_SAMPLES, N_EQUATIONS)
- **funcs** (*List[torch.Tensor]*) – Outputs of the networks after parameterization. There are `len(nets)` entries in total. Each entry is a tensor of shape (N_SAMPLES, N_OUTPUT_UNITS).
- **coords** (*List[torch.Tensor]*) – Inputs to the networks; a.k.a. the spatio-temporal coordinates of the system. There are `N_COORDS` entries in total. Each entry is a tensor of shape (N_SAMPLES, 1).

Returns Additional loss. Must be a `torch.Tensor` of empty shape (scalar).

Return type torch.Tensor

compute_func_val (*net*, *cond*, **coordinates*)

Compute the function value evaluated on the points specified by `coordinates`.

Parameters

- **net** (*torch.nn.Module*) – The network to be parameterized and evaluated.
- **cond** (*neurodiffeq.conditions.BaseCondition*) – The condition (a.k.a. parameterization) for the network.
- **coordinates** (*tuple[torch.Tensor]*) – A tuple of coordinate components, each with shape = (-1, 1).

Returns Function values at the sampled points.

Return type torch.Tensor

fit(*max_epochs*, *callbacks=()*, *tqdm_file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*, ***kwargs*)
Run multiple epochs of training and validation, update best loss at the end of each epoch.

If `callbacks` is passed, callbacks are run, one at a time, after training, validating and updating best model.

Parameters

- **max_epochs** (*int*) – Number of epochs to run.
- **callbacks** – A list of callback functions. Each function should accept the `solver` instance itself as its **only** argument.
- **tqdm_file** (*io.StringIO or _io.TextIOWrapper*) – File to write tqdm progress bar. If set to None, tqdm is not used at all. Defaults to `sys.stderr`.

Rtype callbacks list[callable]

Note:

1. This method does not return solution, which is done in the `.get_solution()` method.
2. A callback `cb(solver)` can set `solver._stop_training` to True to perform early stopping.

---

get_internals(*var_names=None*, *return_type='list'*)
Return internal variable(s) of the solver

- If var_names == 'all', return all internal variables as a dict.
- If var_names is single str, return the corresponding variables.
- If var_names is a list and return_type == 'list', return corresponding internal variables as a list.
- If var_names is a list and return_type == 'dict', return a dict with keys in var_names.

Parameters

- **var_names** (*str or list[str]*) – An internal variable name or a list of internal variable names.
- **return_type** (*str*) – {'list', 'dict'}; Ignored if `var_names` is a string.

Returns A single variable, or a list/dict of internal variables as indicated above.

Return type list or dict or any

get_residuals(**coords*, *to_numpy=False*, *best=True*, *no_reshape=False*)
Get the residuals of the differential equation (or system of differential equations) evaluated at given points.

Parameters

- **coords** (*tuple[torch.Tensor] or tuple[np.ndarray]*) – The coordinate values where the residual(s) shall be evaluated. If numpy arrays are passed, the method implicitly creates torch tensors with corresponding values.
- **to_numpy** (*bool*) – Whether to return numpy arrays. Defaults to False.
- **best** (*bool*) – If set to False, the network from the most recent epoch will be used to evaluate the residuals. If set to True, the network from the epoch with the lowest validation loss will be used to evaluate the residuals. Defaults to True.
- **no_reshape** (*bool*) – If set to True, no reshaping will be performed on output. Defaults to False.

Returns The residuals evaluated at given points. If there is only one equation in the differential equation system, a single torch tensor (or numpy array) will be returned. If there are multiple equations, a list of torch tensors (or numpy arrays) will be returned. The returned shape will be the same as the first input coordinate, unless *no_reshape* is set to True. Note that the return value will always be torch tensors (even if `coords` are numpy arrays) unless *to_numpy* is explicitly set to True.

Return type list[*torch.Tensor* or *numpy.array*] or *torch.Tensor* or *numpy.array*

get_solution(*copy=True*, *best=True*)

Get a (callable) solution object. See this usage example:

```
solution = solver.get_solution()
point_coords = train_generator.get_examples()
value_at_points = solution(point_coords)
```

Parameters

- **copy** (*bool*) – Whether to make a copy of the networks so that subsequent training doesn't affect the solution; Defaults to True.
- **best** (*bool*) – Whether to return the solution with lowest loss instead of the solution after the last epoch. Defaults to True.

Returns A solution object which can be called. To evaluate the solution on certain points, you should pass the coordinates vector(s) to the returned solution.

Return type *BaseSolution*

global_epoch

Global epoch count, always equal to the length of train loss history.

Returns Number of training epochs that have been run.

Return type int

run_train_epoch()

Run a training epoch, update history, and perform gradient descent.

run_valid_epoch()

Run a validation epoch and update history.

class neurodiffeq.solvers.**SolverSpherical**(*pde_system*, *conditions*, *r_min=None*, *r_max=None*, *nets=None*, *train_generator=None*, *valid_generator=None*, *analytic_solutions=None*, *optimizer=None*, *loss_fn=None*, *n_batches_train=1*, *n_batches_valid=4*, *metrics=None*, *enforcer=None*, *n_output_units=1*, *shuffle=None*, *batch_size=None*)

Bases: *neurodiffeq.solvers.BaseSolver*

A solver class for solving PDEs in spherical coordinates

Parameters

- **pde_system** (*callable*) – The PDE system to solve, which maps a tuple of three coordinates to a tuple of PDE residuals, both the coordinates and PDE residuals must have shape (n_samples, 1).
- **conditions** (list[*neurodiffeq.conditions.BaseCondition*]) – List of boundary conditions for each target function.

- **r_min** (*float, optional*) – Radius for inner boundary ($r_0 > 0$). Ignored if `train_generator` and `valid_generator` are both set.
- **r_max** (*float, optional*) – Radius for outer boundary ($r_1 > r_0$). Ignored if `train_generator` and `valid_generator` are both set.
- **nets** (*list[torch.nn.Module], optional*) – List of neural networks for parameterized solution. If provided, length of `nets` must equal that of `conditions`
- **train_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – Generator for sampling training points, which must provide a `.get_examples()` method and a `.size` field. `train_generator` must be specified if `r_min` and `r_max` are not set.
- **valid_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – Generator for sampling validation points, which must provide a `.get_examples()` method and a `.size` field. `valid_generator` must be specified if `r_min` and `r_max` are not set.
- **analytic_solutions** (*callable, optional*) – Analytical solutions to be compared with neural net solutions. It maps a tuple of three coordinates to a tuple of function values. Output shape should match that of `nets`.
- **optimizer** (`torch.nn.optim.Optimizer`, optional) – Optimizer to be used for training. Defaults to a `torch.optim.Adam` instance that trains on all parameters of `nets`.
- **loss_fn** (str or *torch.nn.moduesl.loss._Loss* or callable) – The loss function used for training.
  - If a str, must be present in the keys of *neurodiffeq.losses._losses*.
  - If a *torch.nn.modules.loss._Loss* instance, just pass the instance.
  - If any other callable, it must map A) a residual tensor (shape *(n_points, n_equations)*), B) a function values tuple (length *n_funcs*, each element a tensor of shape *(n_points, 1)*), and C) a coordinate values tuple (length *n_coords*, each element a tensor of shape *(n_coords, 1)* to a tensor of empty shape (i.e. a scalar). The returned tensor must be connected to the computational graph, so that backpropagation can be performed.
- **n_batches_train** (*int, optional*) – Number of batches to train in every epoch, where batch-size equals `train_generator.size`. Defaults to 1.
- **n_batches_valid** (*int, optional*) – Number of batches to validate in every epoch, where batch-size equals `valid_generator.size`. Defaults to 4.
- **metrics** (*dict, optional*) – Additional metrics to be logged (besides loss). `metrics` should be a dict where
  - Keys are metric names (e.g. 'analytic_mse');
  - Values are functions (callables) that computes the metric value. These functions must accept the same input as the differential equation `diff_eq`.
- **enforcer** (*callable*) – A function of signature `enforcer(net: nn.Module, cond: neurodiffeq.conditions.BaseCondition, coords: Tuple[torch.Tensor]) -> torch.Tensor` that returns the dependent variable value evaluated on the batch.
- **n_output_units** (*int, optional*) – Number of output units for each neural network. Ignored if `nets` is specified. Defaults to 1.
- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify `n_batches_train` and `n_batches_valid` instead.

- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

additional_loss(*residual*, *funcs*, *coords*)

Additional loss terms for training. This method is to be overridden by subclasses. This method can use any of the internal variables: self.nets, self.conditions, self.global_epoch, etc.

Parameters

- **residual** (*torch.Tensor*) – Residual tensor of differential equation. It has shape (N_SAMPLES, N_EQUATIONS)
- **funcs** (*List[torch.Tensor]*) – Outputs of the networks after parameterization. There are `len(nets)` entries in total. Each entry is a tensor of shape (N_SAMPLES, N_OUTPUT_UNITS).
- **coords** (*List[torch.Tensor]*) – Inputs to the networks; a.k.a. the spatio-temporal coordinates of the system. There are `N_COORDS` entries in total. Each entry is a tensor of shape (N_SAMPLES, 1).

Returns Additional loss. Must be a `torch.Tensor` of empty shape (scalar).

Return type torch.Tensor

compute_func_val(*net*, *cond*, **coordinates*)

Enforce condition on network with inputs. If self.enforcer is set, use it. Otherwise, fill cond.enforce() with as many arguments as needed.

Parameters

- **net** (*torch.nn.Module*) – Network for parameterized solution.
- **cond** (*neurodiffeq.conditions.BaseCondition*) – Condition (a.k.a. parameterization) for the network.
- **coordinates** (*tuple[torch.Tensor]*) – A tuple of vectors, each with shape = (-1, 1).

Returns Function values at sampled points.

Return type torch.Tensor

fit(*max_epochs*, *callbacks=()*, *tqdm_file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*, ***kwargs*)

Run multiple epochs of training and validation, update best loss at the end of each epoch.

If `callbacks` is passed, callbacks are run, one at a time, after training, validating and updating best model.

Parameters

- **max_epochs** (*int*) – Number of epochs to run.
- **callbacks** – A list of callback functions. Each function should accept the `solver` instance itself as its **only** argument.
- **tqdm_file** (*io.StringIO or _io.TextIOWrapper*) – File to write tqdm progress bar. If set to None, tqdm is not used at all. Defaults to `sys.stderr`.

Rtype callbacks list[callable]

Note:

1. This method does not return solution, which is done in the `.get_solution()` method.

2. A callback `cb(solver)` can set `solver._stop_training` to True to perform early stopping.

get_internals(*var_names=None*, *return_type='list'*)
Return internal variable(s) of the solver

- If var_names == 'all', return all internal variables as a dict.
- If var_names is single str, return the corresponding variables.
- If var_names is a list and return_type == 'list', return corresponding internal variables as a list.
- If var_names is a list and return_type == 'dict', return a dict with keys in var_names.

Parameters

- **var_names** (*str or list[str]*) – An internal variable name or a list of internal variable names.
- **return_type** (*str*) – {'list', 'dict'}; Ignored if `var_names` is a string.

Returns A single variable, or a list/dict of internal variables as indicated above.

Return type list or dict or any

get_residuals(**coords*, *to_numpy=False*, *best=True*, *no_reshape=False*)
Get the residuals of the differential equation (or system of differential equations) evaluated at given points.

Parameters

- **coords** (*tuple[torch.Tensor] or tuple[np.ndarray]*) – The coordinate values where the residual(s) shall be evaluated. If numpy arrays are passed, the method implicitly creates torch tensors with corresponding values.
- **to_numpy** (*bool*) – Whether to return numpy arrays. Defaults to False.
- **best** (*bool*) – If set to False, the network from the most recent epoch will be used to evaluate the residuals. If set to True, the network from the epoch with the lowest validation loss will be used to evaluate the residuals. Defaults to True.
- **no_reshape** (*bool*) – If set to True, no reshaping will be performed on output. Defaults to False.

Returns The residuals evaluated at given points. If there is only one equation in the differential equation system, a single torch tensor (or numpy array) will be returned. If there are multiple equations, a list of torch tensors (or numpy arrays) will be returned. The returned shape will be the same as the first input coordinate, unless *no_reshape* is set to True. Note that the return value will always be torch tensors (even if `coords` are numpy arrays) unless *to_numpy* is explicitly set to True.

Return type list[*torch.Tensor* or *numpy.array*] or *torch.Tensor* or *numpy.array*

get_solution(*copy=True*, *best=True*, *harmonics_fn=None*)
Get a (callable) solution object. See this usage example:

```
solution = solver.get_solution()
point_coords = train_generator.get_examples()
value_at_points = solution(point_coords)
```

Parameters

- **copy** (*bool*) – Whether to make a copy of the networks so that subsequent training doesn't affect the solution; Defaults to True.

- **best** (*bool*) – Whether to return the solution with lowest loss instead of the solution after the last epoch. Defaults to True.
- **harmonics_fn** (*callable*) – If set, use it as function basis for returned solution.

Returns The solution after training.

Return type neurodiffeq.solvers.BaseSolution

global_epoch
Global epoch count, always equal to the length of train loss history.

Returns Number of training epochs that have been run.

Return type int

run_train_epoch()
Run a training epoch, update history, and perform gradient descent.

run_valid_epoch()
Run a validation epoch and update history.

4.5 neurodiffeq.monitors

class neurodiffeq.monitors.**BaseMonitor**(*check_every=None*)
Bases: abc.ABC

A tool for checking the status of the neural network during training.

A monitor keeps track of a matplotlib.figure.Figure instance and updates the plot whenever its check() method is called (usually by a neurodiffeq.solvers.BaseSolver instance).

Note: Currently, the check() method can only run synchronously. It blocks the training / validation process, so don't call the check() method too often.

---

to_callback(*fig_dir=None*, *format=None*, *logger=None*)
Return a callback that updates the monitor plots, which will be run

1. Every self.check_every epochs; and
2. After the last local epoch.

Parameters

- **fig_dir** (*str*) – Directory for saving monitor figs; if not specified, figs will not be saved.
- **format** (*str*) – Format for saving figures: {'jpg', 'png' (default), ...}.
- **logger** (str or logging.Logger) – The logger (or its name) to be used for the returned callback. Defaults to the 'root' logger.

Returns The callback that updates the monitor plots.

Return type *neurodiffeq.callbacks.BaseCallback*

class neurodiffeq.monitors.**MetricsMonitor**(*check_every=None*)
Bases: *neurodiffeq.monitors.BaseMonitor*

A monitor for visualizing the loss and other metrics. This monitor does not visualize the solution.

Parameters **check_every** (*int, optional*) – The frequency of checking the neural network represented by the number of epochs between two checks. Defaults to 100.

to_callback (*fig_dir=None*, *format=None*, *logger=None*)
Return a callback that updates the monitor plots, which will be run

1. Every `self.check_every` epochs; and
2. After the last local epoch.

Parameters

- **fig_dir** (*str*) – Directory for saving monitor figs; if not specified, figs will not be saved.
- **format** (*str*) – Format for saving figures: {'jpg', 'png' (default), ... }.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for the returned callback. Defaults to the 'root' logger.

Returns The callback that updates the monitor plots.

Return type *neurodiffeq.callbacks.BaseCallback*

class neurodiffeq.monitors.**Monitor1D**(*t_min*, *t_max*, *check_every=None*)
Bases: *neurodiffeq.monitors.BaseMonitor*

A monitor for checking the status of the neural network during training.

Parameters

- **t_min** (*float*) – The lower bound of time domain that we want to monitor.
- **t_max** (*float*) – The upper bound of time domain that we want to monitor.
- **check_every** (*int, optional*) – The frequency of checking the neural network represented by the number of epochs between two checks. Defaults to 100.

check(*nets*, *conditions*, *history*)
Draw 2 plots: One shows the shape of the current solution. The other shows the history training loss and validation loss.

Parameters

- **nets** (list[*torch.nn.Module*]) – The neural networks that approximates the ODE (system).
- **conditions** (list[*neurodiffeq.ode.BaseCondition*]) – The initial/boundary conditions of the ODE (system).
- **history** (*dict['train': list[float], 'valid': list[float]]*) – The history of training loss and validation loss. The 'train_loss' entry is a list of training loss and 'valid_loss' entry is a list of validation loss.

Note: *check* is meant to be called by the function *solve* and *solve_system*.

---

to_callback(*fig_dir=None*, *format=None*, *logger=None*)
Return a callback that updates the monitor plots, which will be run

1. Every `self.check_every` epochs; and
2. After the last local epoch.

Parameters

- **fig_dir** (*str*) – Directory for saving monitor figs; if not specified, figs will not be saved.
- **format** (*str*) – Format for saving figures: {'jpg', 'png' (default), ...}.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for the returned callback. Defaults to the 'root' logger.

Returns The callback that updates the monitor plots.

Return type *neurodiffeq.callbacks.BaseCallback*

class neurodiffeq.monitors.**Monitor2D**(*xy_min*, *xy_max*, *check_every=None*, *valid_generator=None*, *solution_style='heatmap'*, *equal_aspect=True*, *ax_width=5.0*, *ax_height=4.0*, *n_col=2*, *levels=20*)

Bases: *neurodiffeq.monitors.BaseMonitor*

A monitor for checking the status of the neural network during training. The number and layout of subplots (matplotlib axes) will be finalized after the first `.check()` call.

Parameters

- **xy_min** (*tuple[float, float], optional*) – The lower bound of 2 dimensions. If we only care about $x \geq x_0$ and $y \geq y_0$, then *xy_min* is *(x_0, y_0)*.
- **xy_max** (*tuple[float, float], optional*) – The upper bound of 2 dimensions. If we only care about $x \leq x_1$ and $y \leq y_1$, then *xy_min* is *(x_1, y_1)*.
- **check_every** (*int, optional*) – The frequency of checking the neural network represented by the number of epochs between two checks. Defaults to 100.
- **valid_generator** (`neurodiffeq.generators.BaseGenerator`) – The generator used to sample points from the domain when visualizing the solution. The generator is only called once (during instantiating the generator), and its outputs are stored. Defaults to a 32x32 `Generator2D` with method 'equally-spaced'.
- **solution_style** (*str*) –
  - If set to 'heatmap', solution visualization will be a contour heat map of u w.r.t. x and y. Useful when visualizing a 2-D spatial solution.
  - If set to 'curves', solution visualization will be u-x curves instead of a 2d heat map. Each curve corresponds to a t value. Useful when visualizing 1D spatio-temporal solution. The first coordinate is interpreted as x and the second as t.

  Defaults to 'heatmap'.
- **equal_aspect** (*bool*) – Whether to set aspect ratio to 1:1 for heatmap. Defaults to True. Ignored if *solutions_style* is 'curves'.
- **ax_width** (*float*) – Width for each solution visualization. Note that this is different from width for metrics history, which is equal to `ax_width` × `n_cols`.
- **ax_height** (*float*) – Height for each solution visualization and metrics history plot.
- **n_col** (*int*) – Number of solution visualizations to plot in each row. Note there is always only 1 plot for metrics history plot per row.
- **levels** (*int*) – Number of levels to plot with contourf (heatmap). Defaults to 20.

check(*nets*, *conditions*, *history*)

Draw 2 plots: One shows the shape of the current solution (with heat map). The other shows the history training loss and validation loss.

Parameters

- **nets** (list [*torch.nn.Module*]) – The neural networks that approximates the PDE.
- **conditions** (list [*neurodiffeq.conditions.BaseCondition*]) – The initial/boundary condition of the PDE.
- **history** (*dict['train': list[float], 'valid': list[float]]*) – The history of training loss and validation loss. The 'train' entry is a list of training loss and 'valid' entry is a list of validation loss.

Note: *check* is meant to be called by the function *solve2D*.

---

to_callback(*fig_dir=None*, *format=None*, *logger=None*)

Return a callback that updates the monitor plots, which will be run

1. Every `self.check_every` epochs; and
2. After the last local epoch.

Parameters

- **fig_dir** (*str*) – Directory for saving monitor figs; if not specified, figs will not be saved.
- **format** (*str*) – Format for saving figures: {'jpg', 'png' (default), ...}.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for the returned callback. Defaults to the 'root' logger.

Returns The callback that updates the monitor plots.

Return type *neurodiffeq.callbacks.BaseCallback*

class neurodiffeq.monitors.**MonitorSpherical**(*r_min*, *r_max*, *check_every=None*, *var_names=None*, *shape=(10, 10, 10)*, *r_scale='linear'*, *theta_min=0.0*, *theta_max=3.141592653589793*, *phi_min=0.0*, *phi_max=6.283185307179586*)

Bases: *neurodiffeq.monitors.BaseMonitor*

A monitor for checking the status of the neural network during training.

Parameters

- **r_min** (*float*) – The lower bound of radius, i.e., radius of interior boundary.
- **r_max** (*float*) – The upper bound of radius, i.e., radius of exterior boundary.
- **check_every** (*int, optional*) – The frequency of checking the neural network represented by the number of epochs between two checks. Defaults to 100.
- **var_names** (*list[str]*) – Names of dependent variables. If provided, shall be used for plot titles. Defaults to None.
- **shape** (*tuple[int]*) – Shape of mesh for visualizing the solution. Defaults to (10, 10, 10).
- **r_scale** (*str*) – 'linear' or 'log'. Controls the grid point in the r direction. Defaults to 'linear'.
- **theta_min** (*float*) – The lower bound of polar angle. Defaults to 0.

- **theta_max** (*float*) – The upper bound of polar angle. Defaults to π.
- **phi_min** (*float*) – The lower bound of azimuthal angle. Defaults to 0.
- **phi_max** (*float*) – The upper bound of azimuthal angle. Defaults to 2π.

check(*nets*, *conditions*, *history*, *analytic_mse_history=None*)
Draw (3n + 2) plots

1. For each function $u_i(r, \phi, \theta)$, there are 3 axes:
   - one ax for u-r curves grouped by ϕ
   - one ax for u-r curves grouped by θ
   - one ax for u-θ-ϕ contour heat map
2. Additionally, one ax for training and validaiton loss, another for the rest of the metrics

Parameters

- **nets** (list [*torch.nn.Module*]) – The neural networks that approximates the PDE.
- **conditions** (list [*neurodiffeq.conditions.BaseCondition*]) – The initial/boundary condition of the PDE.
- **history** (*dict[str, list[float]]*) – A dict of history of training metrics and validation metrics, where keys are metric names (str) and values are list of metrics values (list[float]). It must contain a 'train_loss' key and a 'valid_loss' key.
- **analytic_mse_history** (*dict['train': list[float], 'valid': list[float]], deprecated*) – **[DEPRECATED]** Include 'train_analytic_mse' and 'valid_analytic_mse' in `history` instead.

Note: `check` is meant to be called by `neurodiffeq.solvers.BaseSolver`.

---

customization()
Customized tweaks can be implemented by overwriting this method.

set_variable_count(*n*)
Manually set the number of scalar fields to be visualized; If not set, defaults to length of `nets` passed to `self.check()` every time `self.check()` is called.

Parameters **n** (*int*) – number of scalar fields to overwrite default

Returns self

to_callback(*fig_dir=None*, *format=None*, *logger=None*)
Return a callback that updates the monitor plots, which will be run

1. Every `self.check_every` epochs; and
2. After the last local epoch.

Parameters

- **fig_dir** (*str*) – Directory for saving monitor figs; if not specified, figs will not be saved.
- **format** (*str*) – Format for saving figures: {'jpg', 'png' (default), ...}.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for the returned callback. Defaults to the 'root' logger.

Returns The callback that updates the monitor plots.

Return type *neurodiffeq.callbacks.BaseCallback*

unset_variable_count()
Manually unset the number of scalar fields to be visualized; Once unset, the number defaults to length of `nets` passed to `self.check()` every time `self.check()` is called.

Returns self

class neurodiffeq.monitors.**MonitorSphericalHarmonics**(*r_min*, *r_max*, *check_every=None*, *var_names=None*, *shape=(10, 10, 10)*, *r_scale='linear'*, *harmonics_fn=None*, *theta_min=0.0*, *theta_max=3.141592653589793*, *phi_min=0.0*, *phi_max=6.283185307179586*, *max_degree=None*)
Bases: *neurodiffeq.monitors.MonitorSpherical*

A monitor for checking the status of the neural network during training.

Parameters

- **r_min** (*float*) – The lower bound of radius, i.e., radius of interior boundary.
- **r_max** (*float*) – The upper bound of radius, i.e., radius of exterior boundary.
- **check_every** (*int, optional*) – The frequency of checking the neural network represented by the number of epochs between two checks. Defaults to 100.
- **var_names** (*list[str]*) – The names of dependent variables; if provided, shall be used for plot titles. Defaults to None
- **shape** (*tuple[int]*) – Shape of mesh for visualizing the solution. Defaults to (10, 10, 10).
- **r_scale** (*str*) – 'linear' or 'log'. Controls the grid point in the r direction. Defaults to 'linear'.
- **harmonics_fn** (*callable*) – A mapping from θ and ϕ to basis functions, e.g., spherical harmonics.
- **theta_min** (*float*) – The lower bound of polar angle. Defaults to 0
- **theta_max** (*float*) – The upper bound of polar angle. Defaults to π.
- **phi_min** (*float*) – The lower bound of azimuthal angle. Defaults to 0.
- **phi_max** (*float*) – The upper bound of azimuthal angle. Defaults to 2π.
- **max_degree** (*int*) – **DEPRECATED and SUPERSEDED** by `harmonics_fn`. Highest used for the harmonic basis.

check(*nets*, *conditions*, *history*, *analytic_mse_history=None*)
Draw (3n + 2) plots

1. For each function $u_i(r, \phi, \theta)$, there are 3 axes:
   - one ax for u-r curves grouped by ϕ
   - one ax for u-r curves grouped by θ

- one ax for u-θ-ϕ contour heat map

2. Additionally, one ax for training and validaiton loss, another for the rest of the metrics

Parameters

- **nets** (list [*torch.nn.Module*]) – The neural networks that approximates the PDE.
- **conditions** (list [*neurodiffeq.conditions.BaseCondition*]) – The initial/boundary condition of the PDE.
- **history** (*dict[str, list[float]]*) – A dict of history of training metrics and validation metrics, where keys are metric names (str) and values are list of metrics values (list[float]). It must contain a 'train_loss' key and a 'valid_loss' key.
- **analytic_mse_history** (*dict['train': list[float], 'valid': list[float]], deprecated*) – **[DEPRECATED]** Include 'train_analytic_mse' and 'valid_analytic_mse' in `history` instead.

Note: `check` is meant to be called by `neurodiffeq.solvers.BaseSolver`.

---

customization()
Customized tweaks can be implemented by overwriting this method.

set_variable_count(*n*)
Manually set the number of scalar fields to be visualized; If not set, defaults to length of `nets` passed to `self.check()` every time `self.check()` is called.

Parameters **n** (*int*) – number of scalar fields to overwrite default

Returns self

to_callback(*fig_dir=None*, *format=None*, *logger=None*)
Return a callback that updates the monitor plots, which will be run

1. Every `self.check_every` epochs; and
2. After the last local epoch.

Parameters

- **fig_dir** (*str*) – Directory for saving monitor figs; if not specified, figs will not be saved.
- **format** (*str*) – Format for saving figures: {'jpg', 'png' (default), ...}.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for the returned callback. Defaults to the 'root' logger.

Returns The callback that updates the monitor plots.

Return type *neurodiffeq.callbacks.BaseCallback*

unset_variable_count()
Manually unset the number of scalar fields to be visualized; Once unset, the number defaults to length of `nets` passed to `self.check()` every time `self.check()` is called.

Returns self

class neurodiffeq.monitors.**StreamPlotMonitor2D**(*xy_min*, *xy_max*, *pairs*, *nx=32*, *ny=32*, *check_every=None*, *mask_fn=None*, *ax_width=13.0*, *ax_height=10.0*, *n_col=2*, *stream_kwargs=None*, *equal_aspect=True*, *field_names=None*)

Bases: *neurodiffeq.monitors.BaseMonitor*

to_callback(*fig_dir=None*, *format=None*, *logger=None*)

Return a callback that updates the monitor plots, which will be run

1. Every `self.check_every` epochs; and
2. After the last local epoch.

Parameters

- **fig_dir** (*str*) – Directory for saving monitor figs; if not specified, figs will not be saved.
- **format** (*str*) – Format for saving figures: {'jpg', 'png' (default), ...}.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for the returned callback. Defaults to the 'root' logger.

Returns The callback that updates the monitor plots.

Return type *neurodiffeq.callbacks.BaseCallback*

4.6 *neurodiffeq.ode*

neurodiffeq.ode.**solve**(*ode*, *condition*, *t_min=None*, *t_max=None*, *net=None*, *train_generator=None*, *valid_generator=None*, *optimizer=None*, *criterion=None*, *n_batches_train=1*, *n_batches_valid=4*, *additional_loss_term=None*, *metrics=None*, *max_epochs=1000*, *monitor=None*, *return_internal=False*, *return_best=False*, *batch_size=None*, *shuffle=None*)

Train a neural network to solve an ODE.

Parameters

- **ode** (*callable*) – The ODE to solve. If the ODE is $F(x,t)=0$ where x is the dependent variable and t is the independent variable, then *ode* should be a function that maps (x,t) to $F(x,t)$.
- **condition** (*neurodiffeq.conditions.BaseCondition*) – The initial/boundary condition.
- **net** (*torch.nn.Module*, optional) – The neural network used to approximate the solution. Defaults to None.
- **t_min** (*float*) – The lower bound of the domain (t) on which the ODE is solved, only needed when train_generator or valid_generator are not specified. Defaults to None
- **t_max** (*float*) – The upper bound of the domain (t) on which the ODE is solved, only needed when train_generator or valid_generator are not specified. Defaults to None
- **train_generator** (*neurodiffeq.generators.Generator1D*, optional) – The example generator to generate 1-D training points. Default to None.
- **valid_generator** (*neurodiffeq.generators.Generator1D*, optional) – The example generator to generate 1-D validation points. Default to None.

- **optimizer** (*torch.optim.Optimizer*, optional) – The optimization method to use for training. Defaults to None.
- **criterion** (*torch.nn.modules.loss._Loss*, optional) – The loss function to use for training. Defaults to None.
- **n_batches_train** (*int, optional*) – Number of batches to train in every epoch, where batch-size equals `train_generator.size`. Defaults to 1.
- **n_batches_valid** (*int, optional*) – Number of batches to validate in every epoch, where batch-size equals `valid_generator.size`. Defaults to 4.
- **additional_loss_term** (*callable*) – Extra terms to add to the loss function besides the part specified by *criterion*. The input of *additional_loss_term* should be the same as *ode*.
- **metrics** (*dict[string, callable]*) – Metrics to keep track of during training. The metrics should be passed as a dictionary where the keys are the names of the metrics, and the values are the corresponding function. The input functions should be the same as *ode* and the output should be a numeric value. The metrics are evaluated on both the training set and validation set.
- **max_epochs** (*int, optional*) – The maximum number of epochs to train. Defaults to 1000.
- **monitor** (*neurodiffeq.ode.Monitor*, optional) – The monitor to check the status of neural network during training. Defaults to None.
- **return_internal** (*bool, optional*) – Whether to return the nets, conditions, training generator, validation generator, optimizer and loss function. Defaults to False.
- **return_best** (*bool, optional*) – Whether to return the nets that achieved the lowest validation loss. Defaults to False.
- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify `n_batches_train` and `n_batches_valid` instead.
- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

Returns The solution of the ODE. The history of training loss and validation loss. Optionally, the nets, conditions, training generator, validation generator, optimizer and loss function.

Return type tuple[*neurodiffeq.ode.Solution*, dict] or tuple[*neurodiffeq.ode.Solution*, dict, dict]

Note: This function is deprecated, use a `neurodiffeq.solvers.Solver1D` instead.

`neurodiffeq.ode.`**solve_system**(*ode_system*, *conditions*, *t_min*, *t_max*, *single_net=None*, *nets=None*, *train_generator=None*, *valid_generator=None*, *optimizer=None*, *criterion=None*, *n_batches_train=1*, *n_batches_valid=4*, *additional_loss_term=None*, *metrics=None*, *max_epochs=1000*, *monitor=None*, *return_internal=False*, *return_best=False*, *batch_size=None*, *shuffle=None*)

Train a neural network to solve an ODE.

Parameters

- **ode_system** (*callable*) – The ODE system to solve. If the ODE system consists of equations $F_i(x_1, x_2, ..., x_n, t) = 0$ where x_i is the dependent i-th variable and t is the independent variable, then *ode_system* should be a function that maps $(x_1, x_2, ..., x_n, t)$ to a list where the i-th entry is $F_i(x_1, x_2, ..., x_n, t)$.

- **conditions** (list[*neurodiffeq.conditions.BaseCondition*]) – The initial/boundary conditions. The ith entry of the conditions is the condition that x_i should satisfy.
- **t_min** (*float.*) – The lower bound of the domain (t) on which the ODE is solved, only needed when train_generator or valid_generator are not specified. Defaults to None
- **t_max** (*float*) – The upper bound of the domain (t) on which the ODE is solved, only needed when train_generator or valid_generator are not specified. Defaults to None.
- **single_net** – The single neural network used to approximate the solution. Only one of *single_net* and *nets* should be specified. Defaults to None
- **single_net** – *torch.nn.Module*, optional
- **nets** (list[*torch.nn.Module*], optional) – The neural networks used to approximate the solution. Defaults to None.
- **train_generator** (*neurodiffeq.generators.Generator1D*, optional) – The example generator to generate 1-D training points. Default to None.
- **valid_generator** (*neurodiffeq.generators.Generator1D*, optional) – The example generator to generate 1-D validation points. Default to None.
- **optimizer** (*torch.optim.Optimizer*, optional) – The optimization method to use for training. Defaults to None.
- **criterion** (*torch.nn.modules.loss._Loss*, optional) – The loss function to use for training. Defaults to None and sum of square of the output of *ode_system* will be used.
- **n_batches_train** (*int, optional*) – Number of batches to train in every epoch, where batch-size equals `train_generator.size`. Defaults to 1.
- **n_batches_valid**(*int, optional*) – Number of batches to validate in every epoch, where batch-size equals `valid_generator.size`. Defaults to 4.
- **additional_loss_term**(*callable*) – Extra terms to add to the loss function besides the part specified by *criterion*. The input of *additional_loss_term* should be the same as *ode_system*.
- **metrics** (*dict[str, callable], optional*) – Additional metrics to be logged (besides loss). `metrics` should be a dict where
  - Keys are metric names (e.g. 'analytic_mse');
  - Values are functions (callables) that computes the metric value. These functions must accept the same input as the differential equation `ode_system`.
- **max_epochs** (*int, optional*) – The maximum number of epochs to train. Defaults to 1000.
- **monitor** (*neurodiffeq.ode.Monitor*, optional) – The monitor to check the status of nerual network during training. Defaults to None.
- **return_internal** (*bool, optional*) – Whether to return the nets, conditions, training generator, validation generator, optimizer and loss function. Defaults to False.
- **return_best** (*bool, optional*) – Whether to return the nets that achieved the lowest validation loss. Defaults to False.
- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify `n_batches_train` and `n_batches_valid` instead.
- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

Returns The solution of the ODE. The history of training loss and validation loss. Optionally, the nets, conditions, training generator, validation generator, optimizer and loss function.

Return type tuple[*neurodiffeq.ode.Solution*, dict] or tuple[*neurodiffeq.ode.Solution*, dict, dict]

Note: This function is deprecated, use a `neurodiffeq.solvers.Solver1D` instead.

4.7 *neurodiffeq.pde*

class neurodiffeq.pde.**CustomBoundaryCondition**(*center_point*, *dirichlet_control_points*, *neumann_control_points=None*)

Bases: *neurodiffeq.conditions.IrregularBoundaryCondition*

A boundary condition with irregular shape.

Parameters

- **center_point** (*pde.Point*) – A point that roughly locate at the center of the domain. It will be used to sort the control points 'clockwise'.
- **dirichlet_control_points** (*list[pde.DirichletControlPoint]*) – a list of points on the Dirichlet boundary

enforce(*net*, **dimensions*)

Enforces this condition on a network.

Parameters

- **net** (*torch.nn.Module*) – The network whose output is to be re-parameterized.
- **coordinates** (*torch.Tensor*) – Inputs of the neural network.

Returns The re-parameterized output, where the condition is automatically satisfied.

Return type *torch.Tensor*

in_domain(**dimensions*)

Given the coordinates (numpy.ndarray), the methods returns an boolean array indicating whether the points lie within the domain.

Parameters **coordinates** (*numpy.ndarray*) – Input tensors, each with shape (n_samples, 1).

Returns Whether each point lies within the domain.

Return type *numpy.ndarray*

Note:

- This method is meant to be used by monitors for irregular domain visualization.

---

parameterize(*output_tensor*, **input_tensors*)

Re-parameterizes output(s) of a network.

Parameters

- **output_tensor** (*torch.Tensor*) – Output of the neural network.
- **input_tensors** (*torch.Tensor*) – Inputs to the neural network; i.e., sampled coordinates; i.e., independent variables.

Returns The re-parameterized output of the network.

Return type *torch.Tensor*

Note: This method is **abstract** for BaseCondition

---

set_impose_on(*ith_unit*)
[DEPRECATED] When training several functions with a single, multi-output network, this method is called (by a *Solver* class or a *solve* function) to keep track of which output is being parameterized.

Parameters **ith_unit** (*int*) – The index of network output to be parameterized.

Note: This method is deprecated and retained for backward compatibility only. Users interested in enforcing conditions on multi-output networks should consider using a `neurodiffeq.conditions.EnsembleCondition`.

---

class neurodiffeq.pde.**DirichletControlPoint**(*loc*, *val*)
Bases: *neurodiffeq.pde.Point*

A 2D point on the Dirichlet boundary.

Parameters

- **loc** (*tuple[float, float]*) – The location of the point in the form of (x, y).
- **val** (*float*) – The expected value of u at this location.($u(x, y)$ is the function we are solving for)

class neurodiffeq.pde.**NeumannControlPoint**(*loc*, *val*, *normal_vector*)
Bases: *neurodiffeq.pde.Point*

A 2D point on the Neumann boundary.

Parameters

- **loc** (*tuple[float, float]*) – The location of the point in the form of (x, y).
- **val** (*float*) – The expected normal derivative of u at this location. ($u(x, y)$ is the function we are solving for)

class neurodiffeq.pde.**Point**(*loc*)
Bases: object

A 2D point.

Parameters **loc** (*tuple[float, float]*) – The location of the point in the form of (x, y).

neurodiffeq.pde.**make_animation**(*solution*, *xs*, *ts*)
Create animation of 1-D time-dependent problems.

Parameters

- **solution** (*callable*) – Solution function returned by *solve2D* (for a 1-D time-dependent problem).
- **xs** (*numpy.array*) – The locations to evaluate solution.
- **ts** (*numpy.array*) – The time points to evaluate solution.

Returns The animation.

Return type *matplotlib.animation.FuncAnimation*

neurodiffeq.pde.**solve2D**(*pde*, *condition*, *xy_min=None*, *xy_max=None*, *net=None*, *train_generator=None*, *valid_generator=None*, *optimizer=None*, *criterion=None*, *n_batches_train=1*, *n_batches_valid=4*, *additional_loss_term=None*, *metrics=None*, *max_epochs=1000*, *monitor=None*, *return_internal=False*, *return_best=False*, *batch_size=None*, *shuffle=None*)

Train a neural network to solve a PDE with 2 independent variables.

Parameters

- **pde** (*callable*) – The PDE to solve. If the PDE is $F(u, x, y) = 0$ where u is the dependent variable and x and y are the independent variables, then *pde* should be a function that maps (u, x, y) to $F(u, x, y)$.
- **condition** (*neurodiffeq.conditions.BaseCondition*) – The initial/boundary condition.
- **xy_min** (*tuple[float, float], optional*) – The lower bound of 2 dimensions. If we only care about $x \geq x_0$ and $y \geq y_0$, then *xy_min* is *(x_0, y_0)*, only needed when `train_generator` and `valid_generator` are not specified. Defaults to None
- **xy_max** (*tuple[float, float], optional*) – The upper bound of 2 dimensions. If we only care about $x \leq x_1$ and $y \leq y_1$, then *xy_min* is *(x_1, y_1)*, only needed when `train_generator` and `valid_generator` are not specified. Defaults to None
- **net** (*torch.nn.Module*, optional) – The neural network used to approximate the solution. Defaults to None.
- **train_generator** (*neurodiffeq.generators.Generator2D*, optional) – The example generator to generate 1-D training points. Default to None.
- **valid_generator** (*neurodiffeq.generators.Generator2D*, optional) – The example generator to generate 1-D validation points. Default to None.
- **optimizer** (*torch.optim.Optimizer*, optional) – The optimization method to use for training. Defaults to None.
- **criterion** (*torch.nn.modules.loss._Loss*, optional) – The loss function to use for training. Defaults to None.
- **additional_loss_term** (*callable*) – Extra terms to add to the loss function besides the part specified by *criterion*. The input of *additional_loss_term* should be the same as *pde_system*.
- **n_batches_train** (*int, optional*) – Number of batches to train in every epoch, where batch-size equals `train_generator.size`. Defaults to 1.
- **n_batches_valid** (*int, optional*) – Number of batches to validate in every epoch, where batch-size equals `valid_generator.size`. Defaults to 4.
- **metrics** (*dict[string, callable]*) – Metrics to keep track of during training. The metrics should be passed as a dictionary where the keys are the names of the metrics, and the values are the corresponding function. The input functions should be the same as *pde* and the output should be a numeric value. The metrics are evaluated on both the training set and validation set.
- **max_epochs** (*int, optional*) – The maximum number of epochs to train. Defaults to 1000.
- **monitor** (*neurodiffeq.pde.Monitor2D*, optional) – The monitor to check the status of neural network during training. Defaults to None.
- **return_internal** (*bool, optional*) – Whether to return the nets, conditions, training generator, validation generator, optimizer and loss function. Defaults to False.

- **return_best** (*bool, optional*) – Whether to return the nets that achieved the lowest validation loss. Defaults to False.
- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify `n_batches_train` and `n_batches_valid` instead.
- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

Returns The solution of the PDE. The history of training loss and validation loss. Optionally, the nets, conditions, training generator, validation generator, optimizer and loss function. The solution is a function that has the signature *solution(xs, ys, as_type)*.

Return type tuple[*neurodiffeq.pde.Solution*, dict] or tuple[*neurodiffeq.pde.Solution*, dict, dict]

Note: This function is deprecated, use a `neurodiffeq.solvers.Solver2D` instead.

`neurodiffeq.pde.`**solve2D_system**(*pde_system*, *conditions*, *xy_min=None*, *xy_max=None*, *single_net=None*, *nets=None*, *train_generator=None*, *valid_generator=None*, *optimizer=None*, *criterion=None*, *n_batches_train=1*, *n_batches_valid=4*, *additional_loss_term=None*, *metrics=None*, *max_epochs=1000*, *monitor=None*, *return_internal=False*, *return_best=False*, *batch_size=None*, *shuffle=None*)

Train a neural network to solve a PDE with 2 independent variables.

Parameters

- **pde_system** (*callable*) – The PDE system to solve. If the PDE is $F_i(u_1, u_2, ..., u_n, x, y) = 0$ where u_i is the i-th dependent variable and x and y are the independent variables, then *pde_system* should be a function that maps $(u_1, u_2, ..., u_n, x, y)$ to a list where the i-th entry is $F_i(u_1, u_2, ..., u_n, x, y)$.
- **conditions** (list[*neurodiffeq.conditions.BaseCondition*]) – The initial/boundary conditions. The ith entry of the conditions is the condition that x_i should satisfy.
- **xy_min** (*tuple[float, float], optional*) – The lower bound of 2 dimensions. If we only care about $x \geq x_0$ and $y \geq y_0$, then *xy_min* is *(x_0, y_0)*. Only needed when train_generator or valid_generator are not specified. Defaults to None
- **xy_max** (*tuple[float, float], optional*) – The upper bound of 2 dimensions. If we only care about $x \leq x_1$ and $y \leq y_1$, then *xy_min* is *(x_1, y_1)*. Only needed when train_generator or valid_generator are not specified. Defaults to None
- **single_net** – The single neural network used to approximate the solution. Only one of *single_net* and *nets* should be specified. Defaults to None
- **single_net** – *torch.nn.Module*, optional
- **nets** (list[*torch.nn.Module*], optional) – The neural networks used to approximate the solution. Defaults to None.
- **train_generator** (*neurodiffeq.generators.Generator2D*, optional) – The example generator to generate 1-D training points. Default to None.
- **valid_generator** (*neurodiffeq.generators.Generator2D*, optional) – The example generator to generate 1-D validation points. Default to None.
- **optimizer** (*torch.optim.Optimizer*, optional) – The optimization method to use for training. Defaults to None.

- **criterion** (*torch.nn.modules.loss._Loss*, optional) – The loss function to use for training. Defaults to None.
- **n_batches_train** (*int, optional*) – Number of batches to train in every epoch, where batch-size equals `train_generator.size`. Defaults to 1.
- **n_batches_valid** (*int, optional*) – Number of batches to validate in every epoch, where batch-size equals `valid_generator.size`. Defaults to 4.
- **additional_loss_term** (*callable*) – Extra terms to add to the loss function besides the part specified by *criterion*. The input of *additional_loss_term* should be the same as *pde_system*.
- **metrics** (*dict[string, callable]*) – Metrics to keep track of during training. The metrics should be passed as a dictionary where the keys are the names of the metrics, and the values are the corresponding function. The input functions should be the same as *pde_system* and the output should be a numeric value. The metrics are evaluated on both the training set and validation set.
- **max_epochs** (*int, optional*) – The maximum number of epochs to train. Defaults to 1000.
- **monitor** (*neurodiffeq.pde.Monitor2D*, optional) – The monitor to check the status of nerual network during training. Defaults to None.
- **return_internal** (*bool, optional*) – Whether to return the nets, conditions, training generator, validation generator, optimizer and loss function. Defaults to False.
- **return_best** (*bool, optional*) – Whether to return the nets that achieved the lowest validation loss. Defaults to False.
- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify `n_batches_train` and `n_batches_valid` instead.
- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

Returns The solution of the PDE. The history of training loss and validation loss. Optionally, the nets, conditions, training generator, validation generator, optimizer and loss function. The solution is a function that has the signature *solution(xs, ys, as_type)*.

Return type tuple[*neurodiffeq.pde.Solution*, dict] or tuple[*neurodiffeq.pde.Solution*, dict, dict]

Note: This function is deprecated, use a `neurodiffeq.solvers.Solver2D` instead.

4.8 *neurodiffeq.pde_spherical*

`neurodiffeq.pde_spherical.`**solve_spherical**(*pde*, *condition*, *r_min=None*, *r_max=None*, *net=None*, *train_generator=None*, *valid_generator=None*, *analytic_solution=None*, *optimizer=None*, *criterion=None*, *max_epochs=1000*, *monitor=None*, *return_internal=False*, *return_best=False*, *harmonics_fn=None*, *batch_size=None*, *shuffle=None*)

[**DEPRECATED**, use SphericalSolver class instead] Train a neural network to solve one PDE with spherical inputs in 3D space.

Parameters

- **pde** (*callable*) – The PDE to solve. If the PDE is $F(u, r, \theta, \phi) = 0$, where u is the dependent variable and r, θ and ϕ are the independent variables, then *pde* should be a function that maps (u, r, θ, ϕ) to $F(u, r, \theta, \phi)$.
- **condition** (*neurodiffeq.conditions.BaseCondition*) – The initial/boundary condition that u should satisfy.
- **r_min** (*float, optional*) – Radius for inner boundary; ignored if both generators are provided.
- **r_max** (*float, optional*) – Radius for outer boundary; ignored if both generators are provided.
- **net** (*torch.nn.Module*, optional) – The neural network used to approximate the solution. Defaults to None.
- **train_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – The example generator to generate 3-D training points. Default to None.
- **valid_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – The example generator to generate 3-D validation points. Default to None.
- **analytic_solution** (*callable*) – Analytic solution to the pde system, used for testing purposes. It should map (`rs`, `thetas`, `phis`) to u.
- **optimizer** (*torch.optim.Optimizer*, optional) – The optimization method to use for training. Defaults to None.
- **criterion** (*torch.nn.modules.loss._Loss*, optional) – The loss function to use for training. Defaults to None.
- **max_epochs** (*int, optional*) – The maximum number of epochs to train. Defaults to 1000.
- **monitor** (*neurodiffeq.pde_spherical.MonitorSpherical*, optional) – The monitor to check the status of neural network during training. Defaults to None.
- **return_internal** (*bool, optional*) – Whether to return the nets, conditions, training generator, validation generator, optimizer and loss function. Defaults to False.
- **return_best** (*bool, optional*) – Whether to return the nets that achieved the lowest validation loss. Defaults to False.
- **harmonics_fn** (*callable*) – Function basis (spherical harmonics for example) if solving coefficients of a function basis. Used when returning the solution.
- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify `n_batches_train` and `n_batches_valid` instead.
- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

Returns The solution of the PDE. The history of training loss and validation loss. Optionally, MSE against analytic solution, the nets, conditions, training generator, validation generator, optimizer and loss function. The solution is a function that has the signature *solution(xs, ys, as_type)*.

Return type tuple[*neurodiffeq.pde_spherical.SolutionSpherical*, dict] or tuple[*neurodiffeq.pde_spherical.SolutionSpherical*, dict, dict]

Note: This function is deprecated, use a `neurodiffeq.solvers.SphericalSolver` instead

neurodiffeq.pde_spherical.**solve_spherical_system**(*pde_system*, *conditions*, *r_min=None*, *r_max=None*, *nets=None*, *train_generator=None*, *valid_generator=None*, *analytic_solutions=None*, *optimizer=None*, *criterion=None*, *max_epochs=1000*, *monitor=None*, *return_internal=False*, *return_best=False*, *harmonics_fn=None*, *batch_size=None*, *shuffle=None*)

[**DEPRECATED**, use SphericalSolver class instead] Train a neural network to solve a PDE system with spherical inputs in 3D space

Parameters

- **pde_system** (*callable*) – The PDEs ystem to solve. If the PDE is $F_i(u_1, u_2, ..., u_n, r, \theta, \phi) = 0$ where u_i is the i-th dependent variable and r, θ and ϕ are the independent variables, then *pde_system* should be a function that maps $(u_1, u_2, ..., u_n, r, \theta, \phi)$ to a list where the i-th entry is $F_i(u_1, u_2, ..., u_n, r, \theta, \phi)$.
- **conditions** (list[*neurodiffeq.conditions.BaseCondition*]) – The initial/boundary conditions. The ith entry of the conditions is the condition that u_i should satisfy.
- **r_min** (*float, optional*) – Radius for inner boundary. Ignored if both generators are provided.
- **r_max** (*float, optional*) – Radius for outer boundary. Ignored if both generators are provided.
- **nets** (list[*torch.nn.Module*], optional) – The neural networks used to approximate the solution. Defaults to None.
- **train_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – The example generator to generate 3-D training points. Default to None.
- **valid_generator** (*neurodiffeq.generators.BaseGenerator*, optional) – The example generator to generate 3-D validation points. Default to None.
- **analytic_solutions** (*callable*) – Analytic solution to the pde system, used for testing purposes. It should map (rs, thetas, phis) to a list of [u_1, u_2, . . . , u_n].
- **optimizer** (*torch.optim.Optimizer*, optional) – The optimization method to use for training. Defaults to None.
- **criterion** (*torch.nn.modules.loss._Loss*, optional) – The loss function to use for training. Defaults to None.
- **max_epochs** (*int, optional*) – The maximum number of epochs to train. Defaults to 1000.
- **monitor** (*neurodiffeq.pde_spherical.MonitorSpherical*, optional) – The monitor to check the status of neural network during training. Defaults to None.
- **return_internal** (*bool, optional*) – Whether to return the nets, conditions, training generator, validation generator, optimizer and loss function. Defaults to False.
- **return_best** (*bool, optional*) – Whether to return the nets that achieved the lowest validation loss. Defaults to False.
- **harmonics_fn** (*callable*) – Function basis (spherical harmonics for example) if solving coefficients of a function basis. Used when returning solution.

- **batch_size** (*int*) – **[DEPRECATED and IGNORED]** Each batch will use all samples generated. Please specify n_batches_train and n_batches_valid instead.
- **shuffle** (*bool*) – **[DEPRECATED and IGNORED]** Shuffling should be performed by generators.

Returns The solution of the PDE. The history of training loss and validation loss. Optionally, MSE against analytic solutions, the nets, conditions, training generator, validation generator, optimizer and loss function. The solution is a function that has the signature *solution(xs, ys, as_type)*.

Return type tuple[*neurodiffeq.pde_spherical.SolutionSpherical*, dict] or tuple[*neurodiffeq.pde_spherical.SolutionSpherical*, dict, dict]

Note: This function is deprecated, use a `neurodiffeq.solvers.SphericalSolver` instead

4.9 *neurodiffeq.temporal*

class neurodiffeq.temporal.**Approximator**
Bases: abc.ABC

The base class of approximators. An approximator is an approximation of the differential equation's solution. It knows the parameters in the neural network, and how to calculate the loss function and the metrics.

class neurodiffeq.temporal.**BoundaryCondition**(*form*, *points_generator*)
Bases: object

A boundary condition. It is used to initialize `temporal.Approximators`.

Parameters

- **form** (*callable*) – The form of the boundary condition.
  - For a 1D time-dependent problem, if the boundary condition demands that $B(u, x) = 0$, then `form` should be a function that maps u, x, t to $B(u, x)$.
  - For a 2D steady-state problem, if the boundary condition demands that $B(u, x, y) = 0$, then `form` should be a function that maps u, x, y to $B(u, x, y)$.
  - For a 2D steady-state system, if the boundary condition demands that $B(u_i, x, y) = 0$, then `form` should be a function that maps $u_1, u_2, ..., u_n, x, y$ to $B(u_i, x, y)$.
  - For a 2D time-dependent problem, if the boundary condition demands that $B(u, x, y) = 0$, then `form` should be a function that maps u, x, y, t to $B(u_i, x, y)$.

  Basically the function signature of `form` should be the same as the `pde` function of the given `temporal.Approximator`.
- **points_generator** – A generator that generates points on the boundary. It can be a *temporal.generator_1dspatial*, *temporal.generator_2dspatial_segment*, or a generator written by user.

class neurodiffeq.temporal.**FirstOrderInitialCondition**(*u0*)
Bases: object

A first order initial condition. It is used to initialize `temporal.Approximators`.

Parameters **u0** (*callable*) – A function representing the initial condition. If we are solving for u, then *u0* is $u\Big|_{t=0}$. The input of the function depends on where it is used.

- If it is used as the input for *temporal.SingleNetworkApproximator1DSpatialTemporal*, then *u0* should map x to $u(x,t)\Big|_{t=0}$.
- If it is used as the input for *temporal.SingleNetworkApproximator2DSpatialTemporal*, then *u0* should map (x,y) to $u(x,y,t)\Big|_{t=0}$.

class neurodiffeq.temporal.**Monitor1DSpatialTemporal**(*check_on_x*, *check_on_t*, *check_every*)

Bases: object

A monitor for 1D time-dependent problems.

class neurodiffeq.temporal.**Monitor2DSpatial**(*check_on_x*, *check_on_y*, *check_every*)

Bases: object

A Monitor for 2D steady-state problems

class neurodiffeq.temporal.**Monitor2DSpatialTemporal**(*check_on_x*, *check_on_y*, *check_on_t*, *check_every*)

Bases: object

A monitor for 2D time-dependent problems.

class neurodiffeq.temporal.**MonitorMinimal**(*check_every*)

Bases: object

A monitor that shows the loss function and custom metrics.

class neurodiffeq.temporal.**SecondOrderInitialCondition**(*u0*, *u0dot*)

Bases: object

A second order initial condition. It is used to initialize `temporal.Approximators`.

Parameters

- **u0** (*callable*) – A function representing the initial condition. If we are solving for is u, then `u0` is $u\Big|_{t=0}$. The input of the function dependes on where it is used.
  - If it is used as the input for `temporal.SingleNetworkApproximator1DSpatialTemporal`, then `u0` should map x to $u(x,t)\Big|_{t=0}$.
  - If it is used as the input for `temporal.SingleNetworkApproximator2DSpatialTemporal`, then `u0` should map (x,y) to $u(x,y,t)\Big|_{t=0}$.
- **u0dot** (*callable*) – A function representing the initial derivative w.r.t. time. If we are solving for is u, then `u0dot` is $\dfrac{\partial u}{\partial t}\Big|_{t=0}$. The input of the function depends on where it is used.
  - If it is used as the input for `temporal.SingleNetworkApproximator1DSpatialTemporal`, then `u0` should map x to $\dfrac{\partial u}{\partial t}\Big|_{t=0}$.

– If it is used as the input for *temporal.SingleNetworkApproximator2DSpatialTemporal*, then `u0` should map (x, y) to $\left.\frac{\partial u}{\partial t}\right|_{t=0}$.

class neurodiffeq.temporal.**SingleNetworkApproximator1DSpatialTemporal**(*single_network*, *pde*, *initial_condition*, *boundary_conditions*, *boundary_strictness=1.0*)

Bases: *neurodiffeq.temporal.Approximator*

An approximator to approximate the solution of a 1D time-dependent problem. The boundary condition will be enforced by a regularization term in the loss function and the initial condition will be enforced by transforming the output of the neural network.

Parameters

- **single_network** (*torch.nn.Module*) – A neural network with 2 input nodes (x, t) and 1 output node.
- **pde** (*function*) – The PDE to solve. If the PDE is $F(u, x, t) = 0$ then *pde* should be a function that maps (u, x, t) to $F(u, x, t)$.
- **initial_condition** (*temporal.FirstOrderInitialCondition*) – A first order initial condition.
- **boundary_conditions** (list[*temporal.BoundaryCondition*]) – A list of boundary conditions.
- **boundary_strictness** (*float*) – The regularization parameter, defaults to 1. a larger regularization parameter enforces the boundary conditions more strictly.

class neurodiffeq.temporal.**SingleNetworkApproximator2DSpatial**(*single_network*, *pde*, *boundary_conditions*, *boundary_strictness=1.0*)

Bases: *neurodiffeq.temporal.Approximator*

An approximator to approximate the solution of a 2D steady-state problem. The boundary condition will be enforced by a regularization term in the loss function.

Parameters

- **single_network** (*torch.nn.Module*) – A neural network with 2 input nodes (x, y) and 1 output node.
- **pde** (*function*) – The PDE to solve. If the PDE is $F(u, x, y) = 0$ then *pde* should be a function that maps (u, x, y) to $F(u, x, y)$.
- **boundary_conditions** (list[*temporal.BoundaryCondition*]) – A list of boundary conditions.
- **boundary_strictness** (*float*) – The regularization parameter, defaults to 1. A larger regularization parameter enforces the boundary conditions more strictly.

class neurodiffeq.temporal.**SingleNetworkApproximator2DSpatialSystem**(*single_network*, *pde*, *boundary_conditions*, *boundary_strictness=1.0*)

Bases: *neurodiffeq.temporal.Approximator*

An approximator to approximate the solution of a 2D steady-state differential equation system. The boundary condition will be enforced by a regularization term in the loss function.

Parameters

- **single_network** (*torch.nn.Module*) – A neural network with 2 input nodes (x, y) and n output node (n is the number of dependent variables in the differential equation system)
- **pde** (*callable*) – The PDE system to solve. If the PDE is $F_i(u_1, u_2, ..., u_n, x, y) = 0$ where u_i is the i-th dependent variable, then *pde* should be a function that maps $(u_1, u_2, ..., u_n, x, y)$ to a list where the i-th entry is $F_i(u_1, u_2, ..., u_n, x, y)$.
- **boundary_conditions** (list[*temporal.BoundaryCondition*]) – A list of boundary conditions
- **boundary_strictness** (*float*) – The regularization parameter, defaults to 1. a larger regularization parameter enforces the boundary conditions more strictly.

class neurodiffeq.temporal.**SingleNetworkApproximator2DSpatialTemporal**(*single_network*, *pde*, *initial_condition*, *boundary_conditions*, *boundary_strictness=1.0*)

Bases: *neurodiffeq.temporal.Approximator*

An approximator to approximate the solution of a 2D time-dependent problem. The boundary condition will be enforced by a regularization term in the loss function and the initial condition will be enforced by transforming the output of the neural network.

Parameters

- **single_network** (*torch.nn.Module*) – A neural network with 3 input nodes (x, y, t) and 1 output node.
- **pde** (*callable*) – The PDE system to solve. If the PDE is $F(u, x, y, t) = 0$ then *pde* should be a function that maps (u, x, y, t) to $F(u, x, y, t)$.
- **initial_condition** (*temporal.FirstOrderInitialCondition* or *temporal.SecondOrderInitialCondition*) – A first order initial condition.
- **boundary_conditions** (list[*temporal.BoundaryCondition*]) – A list of boundary conditions.
- **boundary_strictness** (*float*) – The regularization parameter, defaults to 1. a larger regularization parameter enforces the boundary conditions more strictly.

neurodiffeq.temporal.**generator_1dspatial**(*size*, *x_min*, *x_max*, *random=True*)

Return a generator that generates 1D points range from x_min to x_max

Parameters

- **size** (*int*) – Number of points to generated when __next__ is invoked.

- **x_min** (*float*) – Lower bound of x.
- **x_max** (*float*) – Upper bound of x.
- **random** (*bool*) –
  - If set to False, then return equally spaced points range from `x_min` to `x_max`.
  - If set to True then generate points randomly.

  Defaults to True.

neurodiffeq.temporal.**generator_2dspatial_rectangle**(*size*, *x_min*, *x_max*, *y_min*, *y_max*, *random=True*)

Return a generator that generates 2D points in a rectangle.

Parameters

- **size** (*int*) – Number of points to generated when *__next__* is invoked.
- **start** (*tuple[float, float]*) – The starting point of the line segment.
- **end** (*tuple[float, float]*) – The ending point of the line segment.
- **random** (*bool*) –
  - If set to False, then return eqally spaced points range from *start* to *end*.
  - If set to Rrue then generate points randomly.

  Defaults to True.

neurodiffeq.temporal.**generator_2dspatial_segment**(*size*, *start*, *end*, *random=True*)

Return a generator that generates 2D points in a line segment.

Parameters

- **size** (*int*) – Number of points to generated when *__next__* is invoked.
- **x_min** (*float*) – Lower bound of x.
- **x_max** (*float*) – Upper bound of x.
- **y_min** (*float*) – Lower bound of y.
- **y_max** (*float*) – Upper bound of y.
- **random** (*bool*) –
  - If set to False, then return a grid where the points are equally spaced in the x and y dimension.
  - If set to True then generate points randomly.

  Defaults to True.

neurodiffeq.temporal.**generator_temporal**(*size*, *t_min*, *t_max*, *random=True*)

Return a generator that generates 1D points range from t_min to t_max

Parameters

- **size** (*int*) – Number of points to generated when *__next__* is invoked.
- **t_min** (*float*) – Lower bound of t.
- **t_max** (*float*) – Upper bound of t.
- **random** (*bool*) –
  - If set to False, then return eqally spaced points range from t_min to t_max.

– If set to True then generate points randomly.

Defaults to True

4.10 *neurodiffeq.function_basis*

class neurodiffeq.function_basis.**BasisOperator**
Bases: abc.ABC

class neurodiffeq.function_basis.**CustomBasis**(*fns*)
Bases: *neurodiffeq.function_basis.FunctionBasis*

class neurodiffeq.function_basis.**FourierLaplacian**(*max_degree=12*)
Bases: *neurodiffeq.function_basis.BasisOperator*

A Laplacian operator (in polar coordinates) acting on $\sum_i R_i(r)F(\phi)$ where F is a Fourier component

Parameters **max_degree** (*int*) – highest degree for the fourier series

class neurodiffeq.function_basis.**FunctionBasis**
Bases: abc.ABC

class neurodiffeq.function_basis.**HarmonicsLaplacian**(*max_degree=4*)
Bases: *neurodiffeq.function_basis.BasisOperator*

Laplacian of spherical harmonics can be reduced in the following way. Using this method, we can avoid the $\frac{1}{\sin\theta}$ singularity

$$
\begin{aligned}
&\nabla^2 R_{l,m}(r)Y_{l,m}(\theta,\phi) \\
&= \left(\nabla_r^2 + \nabla_\theta^2 + \nabla_\phi^2\right)\left(R_{l,m}(r)Y_{l,m}(\theta,\phi)\right) \\
&= Y_{l,m}\nabla_r^2 R_{l,m} + R_{l,m}\left(\left(\nabla_\theta^2 + \nabla_\phi^2\right)Y_{l,m}\right) \\
&= Y_{l,m}\nabla_r^2 R_{l,m} + R_{l,m}\frac{-l(l+1)}{r^2}Y_{l,m} \\
&= Y_{l,m}\left(\nabla_r^2 R_{l,m} + \frac{-l(l+1)}{r^2}R_{l,m}\right)
\end{aligned}
$$

class neurodiffeq.function_basis.**LegendreBasis**(*max_degree*)
Bases: *neurodiffeq.function_basis.FunctionBasis*

class neurodiffeq.function_basis.**RealFourierSeries**(*max_degree=12*)
Bases: *neurodiffeq.function_basis.FunctionBasis*

Real Fourier Series.

Parameters **max_degree** (*int*) – highest degree for the fourier series

class neurodiffeq.function_basis.**RealSphericalHarmonics**(*max_degree=4*)
Bases: *neurodiffeq.function_basis.FunctionBasis*

Spherical harmonics as a function basis

Parameters **max_degree** (*int*) – highest degree (currently only supports l<=4) for the spherical harmonics_fn

neurodiffeq.function_basis.**ZeroOrderSphericalHarmonics**(*max_degree=None*, *degrees=None*)

Zonal harmonics (spherical harmonics with order=0)

Parameters

- **max_degree** (*int*) – highest degrees to be included; degrees will contain {0, 1, ..., max_degree}; ignored if *degrees* is passed
- **degrees** (*list[int]*) – a list of degrees to be used, must be nonnegative and unique; if passed, *max_degrees* will be ignored

neurodiffeq.function_basis.**ZeroOrderSphericalHarmonicsLaplacian**(*max_degree=None*, *degrees=None*)

Laplacian operator acting on coefficients of zonal harmonics (spherical harmonics with order=0)

Parameters

- **max_degree** (*int*) – highest degrees to be included; degrees will contain {0, 1, ..., max_degree}; ignored if *degrees* is passed
- **degrees** (*list[int]*) – a list of degrees to be used, must be nonnegative and unique; if passed, *max_degrees* will be ignored

class neurodiffeq.function_basis.**ZonalSphericalHarmonics**(*max_degree=None*, *degrees=None*)

Bases: *neurodiffeq.function_basis.FunctionBasis*

Zonal harmonics (spherical harmonics with order=0)

Parameters

- **max_degree** (*int*) – highest degrees to be included; degrees will contain {0, 1, ..., max_degree}; ignored if *degrees* is passed
- **degrees** (*list[int]*) – a list of degrees to be used, must be nonnegative and unique; if passed, *max_degrees* will be ignored

class neurodiffeq.function_basis.**ZonalSphericalHarmonicsLaplacian**(*max_degree=None*, *degrees=None*)

Bases: *neurodiffeq.function_basis.BasisOperator*

Laplacian operator acting on coefficients of zonal harmonics (spherical harmonics with order=0)

Parameters

- **max_degree** (*int*) – highest degrees to be included; degrees will contain {0, 1, ..., max_degree}; ignored if *degrees* is passed
- **degrees** (*list[int]*) – a list of degrees to be used, must be nonnegative and unique; if passed, *max_degrees* will be ignored

4.11 *neurodiffeq.generators*

This module contains atomic generator classes and useful tools to construct complex generators out of atomic ones

class neurodiffeq.generators.**BaseGenerator**

Bases: object

Base class for all generators; Children classes must implement a *.get_examples* method and a *.size* field.

class neurodiffeq.generators.**BatchGenerator**(*generator*, *batch_size*)

Bases: *neurodiffeq.generators.BaseGenerator*

A generator which caches samples and returns a single batch of the samples at a time

Parameters

- **generator** (BaseGenerator) – A generator used for getting (cached) examples.
- **batch_size** (*int*) – Number of batches to be returned. It can be larger than size of `generator`, but inefficient if so.

class neurodiffeq.generators.**ConcatGenerator**(**generators*)
Bases: *neurodiffeq.generators.BaseGenerator*

An concatenated generator for sampling points, whose `get_examples()` method returns the concatenated vector of the samples returned by its sub-generators.

Parameters **generators** (*Tuple[BaseGenerator]*) – a sequence of sub-generators, must have a `.size` field and a `.get_examples()` method

---

Note: Not to be confused with `EnsembleGenerator` which returns all the samples of its sub-generators.

---

class neurodiffeq.generators.**EnsembleGenerator**(**generators*)
Bases: *neurodiffeq.generators.BaseGenerator*

A generator for sampling points whose *get_examples* method returns all the samples of its sub-generators. All sub-generator must return tensors of the same shape. The number of tensors returned by each sub-generator can be different.

Parameters **generators** (*Tuple[BaseGenerator]*) – a sequence of sub-generators, must have a .size field and a .get_examples() method

---

Note: Not to be confused with `ConcatGenerator` which returns the concatenated vector of samples returned by its sub-generators.

---

class neurodiffeq.generators.**FilterGenerator**(*generator*, *filter_fn*, *size=None*, *update_size=True*)
Bases: *neurodiffeq.generators.BaseGenerator*

A generator which applies some filtering before samples are returned

Parameters

- **generator** (BaseGenerator) – A generator used to generate samples to be filtered.
- **filter_fn** (*callable*) – A filter to be applied on the sample vectors; maps a list of tensors to a mask tensor.
- **size** (*int*) – Size to be used for *self.size*. If not given, this attribute is initialized to the size of `generator`.
- **update_size** (*bool*) – Whether or not to update *.size* after each call of *self.get_examples*. Defaults to True.

class neurodiffeq.generators.**Generator1D**(*size*, *t_min=0.0*, *t_max=1.0*, *method='uniform'*, *noise_std=None*)
Bases: *neurodiffeq.generators.BaseGenerator*

An example generator for generating 1-D training points.

Parameters

- **size** (*int*) – The number of points to generate each time *get_examples* is called.
- **t_min** (*float, optional*) – The lower bound of the 1-D points generated, defaults to 0.0.

- **t_max** (*float, optional*) – The upper boound of the 1-D points generated, defaults to 1.0.
- **method** (*str, optional*) – The distribution of the 1-D points generated.
  - If set to 'uniform', the points will be drew from a uniform distribution Unif(t_min, t_max).
  - If set to 'equally-spaced', the points will be fixed to a set of linearly-spaced points that go from t_min to t_max.
  - If set to 'equally-spaced-noisy', a normal noise will be added to the previously mentioned set of points.
  - If set to 'log-spaced', the points will be fixed to a set of log-spaced points that go from t_min to t_max.
  - If set to 'log-spaced-noisy', a normal noise will be added to the previously mentioned set of points,
  - If set to 'chebyshev1' or 'chebyshev', the points are chebyshev nodes of the first kind over (t_min, t_max).
  - If set to 'chebyshev2', the points will be chebyshev nodes of the second kind over [t_min, t_max].

  defaults to 'uniform'.

Raises **ValueError** – When provided with an unknown method.

class neurodiffeq.generators.**Generator2D**(*grid=(10, 10)*, *xy_min=(0.0, 0.0)*, *xy_max=(1.0, 1.0)*, *method='equally-spaced-noisy'*, *xy_noise_std=None*)

Bases: *neurodiffeq.generators.BaseGenerator*

An example generator for generating 2-D training points.

Parameters

- **grid** (*tuple[int, int], optional*) – The discretization of the 2 dimensions. If we want to generate points on a $m \times n$ grid, then *grid* is *(m, n)*. Defaults to *(10, 10)*.
- **xy_min** (*tuple[float, float], optional*) – The lower bound of 2 dimensions. If we only care about $x \geq x_0$ and $y \geq y_0$, then *xy_min* is *(x_0, y_0)*. Defaults to *(0.0, 0.0)*.
- **xy_max** (*tuple[float, float], optional*) – The upper boound of 2 dimensions. If we only care about $x \leq x_1$ and $y \leq y_1$, then *xy_min* is *(x_1, y_1)*. Defaults to *(1.0, 1.0)*.
- **method** (*str, optional*) – The distribution of the 2-D points generated.
  - If set to 'equally-spaced', the points will be fixed to the grid specified.
  - If set to 'equally-spaced-noisy', a normal noise will be added to the previously mentioned set of points.
  - If set to 'chebyshev' or 'chebyshev1', the points will be 2-D chebyshev points of the first kind.
  - If set to 'chebyshev2', the points will be 2-D chebyshev points of the second kind.

  Defaults to 'equally-spaced-noisy'.
- **xy_noise_std** (*tuple[int, int], optional, defaults to None*) – The standard deviation of the noise on the x and y dimension. If not specified, the default

value will be (`grid step size on x dimension` / 4, `grid step size on y dimension` / 4).

Raises **ValueError** – When provided with an unknown method.

class neurodiffeq.generators.**Generator3D**(*grid=(10, 10, 10), xyz_min=(0.0, 0.0, 0.0), xyz_max=(1.0, 1.0, 1.0), method='equally-spaced-noisy'*)

Bases: *neurodiffeq.generators.BaseGenerator*

An example generator for generating 3-D training points. NOT TO BE CONFUSED with *GeneratorSpherical*

Parameters

- **grid** (*tuple[int, int, int], optional*) – The discretization of the 3 dimensions. If we want to generate points on a $m \times n \times k$ grid, then *grid* is *(m, n, k)*, defaults to *(10, 10, 10)*.
- **xyz_min** (*tuple[float, float, float], optional*) – The lower bound of 3 dimensions. If we only care about $x \geq x_0$, $y \geq y_0$, and $z \geq z_0$ then *xyz_min* is (x_0, y_0, z_0). Defaults to *(0.0, 0.0, 0.0)*.
- **xyz_max** (*tuple[float, float, float], optional*) – The upper bound of 3 dimensions. If we only care about $x \leq x_1$, $y \leq y_1$, i and $z \leq z_1$ then *xyz_max* is (x_1, y_1, z_1). Defaults to *(1.0, 1.0, 1.0)*.
- **method** (*str, optional*) – The distribution of the 3-D points generated.
  - If set to 'equally-spaced', the points will be fixed to the grid specified.
  - If set to 'equally-spaced-noisy', a normal noise will be added to the previously mentioned set of points.
  - If set to 'chebyshev' or 'chebyshev1', the points will be 3-D chebyshev points of the first kind.
  - If set to 'chebyshev2', the points will be 3-D chebyshev points of the second kind.

  Defaults to 'equally-spaced-noisy'.

Raises **ValueError** – When provided with an unknown method.

class neurodiffeq.generators.**GeneratorND**(*grid=(10, 10), r_min=(0.0, 0.0), r_max=(1.0, 1.0), methods=['equally-spaced', 'equally-spaced'], noisy=True, r_noise_std=None, **kwargs*)

Bases: *neurodiffeq.generators.BaseGenerator*

An example generator for generating N-D training points.

Parameters

- **grid** (*tuple[int, int, .. , int], or it can be int if N=1, optional*) – The discretization of the N dimensions. If we want to generate points on a $n_1 \times n_2 \times ... \times n_N$ grid, then *grid* is *(n_1, n_2, ... , n_N)*. Defaults to *(10, 10)*.
- **r_min** (*tuple[float, .. , float], or it can be float if N=1, optional*) – The lower bound of N dimensions. If we only care about $r_1 \geq r_1^{min}$, $r_2 \geq r_2^{min}$, ... , and $r_N \geq r_N^{min}$ then *r_min* is *(r_1_min, r_2_min, ... , r_N_min)*. Defaults to *(0.0, 0.0)*.
- **r_max** (*tuple[float, .. , float], or it can be float if N=1, optional*) – The upper boound of N dimensions. If we only care about $r_1 \leq r_1^{max}$,

$r_2 \leq r_2^{max}$, ... , and $r_N \leq r_N^{max}$ then *r_max* is *(r_1_max, r_2_max, ... , r_N_max)*. Defaults to *(1.0, 1.0)*.

- **methods** (*list[str, str, .. , str], or it can be str if N=1, optional*) – The a list of the distributions of each of the 1-D points generated that make the total N-D points.
  - If set to 'uniform', the points will be drew from a uniform distribution Unif(r_min[i], r_max[i]).
  - If set to 'equally-spaced', the points will be fixed to a set of linearly-spaced points that go from r_min[i] to r_max[i].
  - If set to 'log-spaced', the points will be fixed to a set of log-spaced points that go from r_min[i] to r_max[i].
  - If set to 'exp-spaced', the points will be fixed to a set of exp-spaced points that go from r_min[i] to r_max[i].
  - If set to 'chebyshev' or 'chebyshev1', the points will be chebyshev points of the first kind that go from r_min[i] to r_max[i].
  - If set to 'chebyshev2', the points will be chebyshev points of the second kind that go from r_min[i] to r_max[i].

  Defaults to ['equally-spaced', 'equally-spaced'].
- **noisy** (*bool*) – if set to True a normal noise will be added to all of the N sets of points that make the generator. Defaults to True.

Raises **ValueError** – When provided with unknown methods.

class neurodiffeq.generators.**GeneratorSpherical**(*size*, *r_min=0.0*, *r_max=1.0*, *method='equally-spaced-noisy'*)

Bases: *neurodiffeq.generators.BaseGenerator*

A generator for generating points in spherical coordinates.

Parameters

- **size** (*int*) – Number of points in 3-D sphere.
- **r_min** (*float, optional*) – Radius of the interior boundary.
- **r_max** (*float, optional*) – Radius of the exterior boundary.
- **method** (*str, optional*) – The distribution of the 3-D points generated.
  - If set to 'equally-radius-noisy', radius of the points will be drawn from a uniform distribution $r \sim U[r_{min}, r_{max}]$.
  - If set to 'equally-spaced-noisy', squared radius of the points will be drawn from a uniform distribution $r^2 \sim U[r_{min}^2, r_{max}^2]$

  Defaults to 'equally-spaced-noisy'.

Note: Not to be confused with `Generator3D`.

class neurodiffeq.generators.**MeshGenerator**(**generators*)

Bases: *neurodiffeq.generators.BaseGenerator*

A generator for sampling points whose *get_examples* method returns a mesh of the samples of its sub-generators. All sub-generators must return tensors of the same shape, or a tuple of tensors of the same shape. The number of tensors returned by each sub-generator can be different, but the intent behind this class is to create an N

dimensional generator from several 1 dimensional generators, so each input generator should represents one of the dimensions of your problem. An exception is made for using a `MeshGenerator` as one of the inputs of another `MeshGenerator`. In that case the original meshed generators are extracted from the input `MeshGenerator`, and the final mesh is used using those (e.g `MeshGenerator(MeshGenerator(g1, g2), g3)` is equivalent to `MeshGenerator(g1, g2, g3)`, where g1, g2 and g3 are `Generator1D`). This is done to make the use of the ^ infix consistent with the use of the `MeshGenerator` class itself (e.g `MeshGenerator(g1, g2, g3)` is equivalent to g1 ^ g2 ^ g3), where g1, g2 and g3 are `Generator1D`).

Parameters **generators** (*Tuple[BaseGenerator]*) – a sequence of sub-generators, must have a .size field and a .get_examples() method

class neurodiffeq.generators.**PredefinedGenerator**(**xs*)
Bases: *neurodiffeq.generators.BaseGenerator*

A generator for generating points that are fixed and predefined.

Parameters **xs** (Tuple[*torch.Tensor*]) – training points that will be returned

get_examples()
Returns the training points. Points are fixed and predefined.

Returns The predefined training points

Return type tuple[*torch.Tensor*]

class neurodiffeq.generators.**ResampleGenerator**(*generator*, *size=None*, *replacement=False*)
Bases: *neurodiffeq.generators.BaseGenerator*

A generator whose output is shuffled and resampled every time

Parameters

- **generator** (BaseGenerator) – A generator used to generate samples to be shuffled and resampled.
- **size** (*int*) – Size of the shuffled output. Defaults to the size of `generator`.
- **replacement** (*bool*) – Whether to sample with replacement or not. Defaults to False.

class neurodiffeq.generators.**SamplerGenerator**(*generator*)
Bases: *neurodiffeq.generators.BaseGenerator*

class neurodiffeq.generators.**StaticGenerator**(*generator*)
Bases: *neurodiffeq.generators.BaseGenerator*

A generator that returns the same samples every time. The static samples are obtained by the sub-generator at instantiation time.

Parameters **generator** (BaseGenerator) – a generator used to generate the static samples

class neurodiffeq.generators.**TransformGenerator**(*generator*, *transforms=None*, *transform=None*)
Bases: *neurodiffeq.generators.BaseGenerator*

A generator which applies certain transformations on the sample vectors.

Parameters

- **generator** (BaseGenerator) – A generator used to generate samples on which transformations will be applied.
- **transforms** (*list[callable]*) – A list of transformations to be applied on the sample vectors. Identity transformation can be replaced with None

- **transform** (*callable*) – A callable that transforms the output(s) of base generator to another (tuple of) coordinate(s).

4.12 *neurodiffeq.operators*

neurodiffeq.operators.**cartesian_to_cylindrical**(*x*, *y*, *z*)

Convert cartesian coordinates (x, y, z) to cylindrical coordinate (ρ, ϕ, z). The input shapes of x, y, and z must be the same. If the azimuthal angle phi is undefined, the default value will be 0.

Parameters

- **x** (*torch.Tensor*) – The x-component of cartesian coordinates.
- **y** (*torch.Tensor*) – The y-component of cartesian coordinates.
- **z** (*torch.Tensor*) – The z-component of cartesian coordinates.

Returns The ρ-, ϕ-, and z-component in cylindrical coordinates.

Return type tuple[*torch.Tensor*]

neurodiffeq.operators.**cartesian_to_spherical**(*x*, *y*, *z*)

Convert cartesian coordinates (x, y, z) to spherical coordinate (r, θ, ϕ). The input shapes of x, y, and z must be the same. If either the polar angle θ or the azimuthal angle phi is not defined, the default value will be 0.

Parameters

- **x** (*torch.Tensor*) – The x-component of cartesian coordinates.
- **y** (*torch.Tensor*) – The y-component of cartesian coordinates.
- **z** (*torch.Tensor*) – The z-component of cartesian coordinates.

Returns The r-, θ-, and ϕ-component in spherical coordinates.

Return type tuple[*torch.Tensor*]

neurodiffeq.operators.**curl**(*u_x*, *u_y*, *u_z*, *x*, *y*, *z*)

Derives and evaluates the curl of a vector field $\mathbf{u}$ in three dimensional cartesian coordinates.

Parameters

- **u_x** (*torch.Tensor*) – The x-component of the vector field u, must have shape (n_samples, 1).
- **u_y** (*torch.Tensor*) – The y-component of the vector field u, must have shape (n_samples, 1).
- **u_z** (*torch.Tensor*) – The z-component of the vector field u, must have shape (n_samples, 1).
- **x** (*torch.Tensor*) – A vector of x-coordinate values, must have shape (n_samples, 1).
- **y** (*torch.Tensor*) – A vector of y-coordinate values, must have shape (n_samples, 1).
- **z** (*torch.Tensor*) – A vector of z-coordinate values, must have shape (n_samples, 1).

Returns The x, y, and z components of the curl, each with shape (n_samples, 1).

Return type tuple[*torch.Tensor*]

neurodiffeq.operators.**cylindrical_curl**(*u_rho*, *u_phi*, *u_z*, *rho*, *phi*, *z*)

Derives and evaluates the cylindrical curl of a cylindrical vector field u.

Parameters

- **u_rho** (*torch.Tensor*) – The ρ-component of the vector field u, must have shape (n_samples, 1).
- **u_phi** (*torch.Tensor*) – The ϕ-component of the vector field u, must have shape (n_samples, 1).
- **u_z** (*torch.Tensor*) – The z-component of the vector field u, must have shape (n_samples, 1).
- **rho** (*torch.Tensor*) – A vector of ρ-coordinate values, must have shape (n_samples, 1).
- **phi** (*torch.Tensor*) – A vector of ϕ-coordinate values, must have shape (n_samples, 1).
- **z** (*torch.Tensor*) – A vector of z-coordinate values, must have shape (n_samples, 1).

Returns The ρ, ϕ, and z components of the curl, each with shape (n_samples, 1).

Return type tuple[*torch.Tensor*]

neurodiffeq.operators.**cylindrical_div**(*u_rho*, *u_phi*, *u_z*, *rho*, *phi*, *z*)

Derives and evaluates the cylindrical divergence of a cylindrical vector field u.

Parameters

- **u_rho** (*torch.Tensor*) – The ρ-component of the vector field u, must have shape (n_samples, 1).
- **u_phi** (*torch.Tensor*) – The ϕ-component of the vector field u, must have shape (n_samples, 1).
- **u_z** (*torch.Tensor*) – The z-component of the vector field u, must have shape (n_samples, 1).
- **rho** (*torch.Tensor*) – A vector of ρ-coordinate values, must have shape (n_samples, 1).
- **phi** (*torch.Tensor*) – A vector of ϕ-coordinate values, must have shape (n_samples, 1).
- **z** (*torch.Tensor*) – A vector of z-coordinate values, must have shape (n_samples, 1).

Returns The divergence evaluated at (ρ, ϕ, z), with shape (n_samples, 1).

Return type *torch.Tensor*

neurodiffeq.operators.**cylindrical_grad**(*u*, *rho*, *phi*, *z*)

Derives and evaluates the cylindrical gradient of a cylindrical scalar field u.

Parameters

- **u** (*torch.Tensor*) – A scalar field u, must have shape (n_samples, 1).
- **rho** (*torch.Tensor*) – A vector of ρ-coordinate values, must have shape (n_samples, 1).
- **phi** (*torch.Tensor*) – A vector of ϕ-coordinate values, must have shape (n_samples, 1).
- **z** (*torch.Tensor*) – A vector of z-coordinate values, must have shape (n_samples, 1).

Returns The ρ, ϕ, and z components of the gradient, each with shape (n_samples, 1).

Return type tuple[*torch.Tensor*]

neurodiffeq.operators.**cylindrical_laplacian**(*u*, *rho*, *phi*, *z*)

Derives and evaluates the cylindrical laplacian of a cylindrical scalar field u.

Parameters

- **u** (*torch.Tensor*) – A scalar field u, must have shape (n_samples, 1).
- **rho** (*torch.Tensor*) – A vector of ρ-coordinate values, must have shape (n_samples, 1).

- **phi** (*torch.Tensor*) – A vector of ϕ-coordinate values, must have shape (n_samples, 1).
- **z** (*torch.Tensor*) – A vector of z-coordinate values, must have shape (n_samples, 1).

Returns The laplacian evaluated at (ρ, ϕ, z), with shape (n_samples, 1).

Return type *torch.Tensor*

neurodiffeq.operators.**cylindrical_to_cartesian**(*rho*, *phi*, *z*)

Convert cylindrical coordinate (ρ, ϕ, z) to cartesian coordinates (x, y, z). The input shapes of rho, phi, and z must be the same.

Parameters

- **rho** (*torch.Tensor*) – The ρ-component of cylindrical coordinates.
- **phi** (*torch.Tensor*) – The ϕ-component (azimuthal angle) of cylindrical coordinates.
- **z** (*torch.Tensor*) – The z-component of cylindrical coordinates.

Returns The x-, y-, and z-component in cartesian coordinates.

Return type tuple[*torch.Tensor*]

neurodiffeq.operators.**cylindrical_vector_laplacian**(*u_rho*, *u_phi*, *u_z*, *rho*, *phi*, *z*)

Derives and evaluates the cylindrical laplacian of a cylindrical vector field u.

Parameters

- **u_rho** (*torch.Tensor*) – The ρ-component of the vector field u, must have shape (n_samples, 1).
- **u_phi** (*torch.Tensor*) – The ϕ-component of the vector field u, must have shape (n_samples, 1).
- **u_z** (*torch.Tensor*) – The z-component of the vector field u, must have shape (n_samples, 1).
- **rho** (*torch.Tensor*) – A vector of ρ-coordinate values, must have shape (n_samples, 1).
- **phi** (*torch.Tensor*) – A vector of ϕ-coordinate values, must have shape (n_samples, 1).
- **z** (*torch.Tensor*) – A vector of z-coordinate values, must have shape (n_samples, 1).

Returns The laplacian evaluated at (ρ, ϕ, z), with shape (n_samples, 1).

Return type *torch.Tensor*

neurodiffeq.operators.**div**(**us_xs*)

Derives and evaluates the divergence of a n-dimensional vector field $\mathbf{u}$ with respect to $\mathbf{x}$.

Parameters **us_xs** (*torch.Tensor*) – The input must have $2n$ tensors, each of shape (n_samples, 1) with the former n tensors being the entries of u and the latter n tensors being the entries of x.

Returns The divergence evaluated at x, with shape (n_samples, 1).

Return type *torch.Tensor*

neurodiffeq.operators.**grad**(*u*, **xs*)

Gradient of tensor u with respect to a tuple of tensors xs. Given u and $x_1, \ldots, x_n$, the function returns $\frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_n}$

Parameters

- **u** (*torch.Tensor*) – The u described above.
- **xs** (*torch.Tensor*) – The sequence of x_i described above.

Returns A tuple of $\frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_n}$

Return type List[*torch.Tensor*]

neurodiffeq.operators.**laplacian**(*u*, **xs*)

Derives and evaluates the laplacian of a scalar field u with respect to $\mathbf{x} = [x_1, x_2, \ldots]$

Parameters

- **u** (*torch.Tensor*) – A scalar field u, must have shape (n_samples, 1).
- **xs** (*torch.Tensor*) – The sequence of x_i described above. Each with shape (n_samples, 1)

Returns The laplacian of u evaluated at $\mathbf{x}$, with shape (n_samples, 1).

Return type *torch.Tensor*

neurodiffeq.operators.**spherical_curl**(*u_r*, *u_theta*, *u_phi*, *r*, *theta*, *phi*)

Derives and evaluates the spherical curl of a spherical vector field u.

Parameters

- **u_r** (*torch.Tensor*) – The r-component of the vector field u, must have shape (n_samples, 1).
- **u_theta** (*torch.Tensor*) – The θ-component of the vector field u, must have shape (n_samples, 1).
- **u_phi** (*torch.Tensor*) – The ϕ-component of the vector field u, must have shape (n_samples, 1).
- **r** (*torch.Tensor*) – A vector of r-coordinate values, must have shape (n_samples, 1).
- **theta** (*torch.Tensor*) – A vector of θ-coordinate values, must have shape (n_samples, 1).
- **phi** (*torch.Tensor*) – A vector of ϕ-coordinate values, must have shape (n_samples, 1).

Returns The r, θ, and ϕ components of the curl, each with shape (n_samples, 1).

Return type tuple[*torch.Tensor*]

neurodiffeq.operators.**spherical_div**(*u_r*, *u_theta*, *u_phi*, *r*, *theta*, *phi*)

Derives and evaluates the spherical divergence of a spherical vector field u.

Parameters

- **u_r** (*torch.Tensor*) – The r-component of the vector field u, must have shape (n_samples, 1).
- **u_theta** (*torch.Tensor*) – The θ-component of the vector field u, must have shape (n_samples, 1).
- **u_phi** (*torch.Tensor*) – The ϕ-component of the vector field u, must have shape (n_samples, 1).
- **r** (*torch.Tensor*) – A vector of r-coordinate values, must have shape (n_samples, 1).
- **theta** (*torch.Tensor*) – A vector of θ-coordinate values, must have shape (n_samples, 1).
- **phi** (*torch.Tensor*) – A vector of ϕ-coordinate values, must have shape (n_samples, 1).

Returns The divergence evaluated at (r, θ, ϕ), with shape (n_samples, 1).

Return type *torch.Tensor*

neurodiffeq.operators.**spherical_grad**(*u*, *r*, *theta*, *phi*)

Derives and evaluates the spherical gradient of a spherical scalar field u.

Parameters

- **u** (*torch.Tensor*) – A scalar field u, must have shape (n_samples, 1).
- **r** (*torch.Tensor*) – A vector of r-coordinate values, must have shape (n_samples, 1).
- **theta** (*torch.Tensor*) – A vector of θ-coordinate values, must have shape (n_samples, 1).
- **phi** (*torch.Tensor*) – A vector of ϕ-coordinate values, must have shape (n_samples, 1).

Returns The r, θ, and ϕ components of the gradient, each with shape (n_samples, 1).

Return type tuple[*torch.Tensor*]

neurodiffeq.operators.**spherical_laplacian**(*u*, *r*, *theta*, *phi*)

Derives and evaluates the spherical laplacian of a spherical scalar field u.

Parameters

- **u** (*torch.Tensor*) – A scalar field u, must have shape (n_samples, 1).
- **r** (*torch.Tensor*) – A vector of r-coordinate values, must have shape (n_samples, 1).
- **theta** (*torch.Tensor*) – A vector of θ-coordinate values, must have shape (n_samples, 1).
- **phi** (*torch.Tensor*) – A vector of ϕ-coordinate values, must have shape (n_samples, 1).

Returns The laplacian evaluated at (r, θ, ϕ), with shape (n_samples, 1).

Return type *torch.Tensor*

neurodiffeq.operators.**spherical_to_cartesian**(*r*, *theta*, *phi*)

Convert spherical coordinate (r, θ, ϕ) to cartesian coordinates (x, y, z). The input shapes of r, theta, and phi must be the same.

Parameters

- **r** (*torch.Tensor*) – The r-component of spherical coordinates.
- **theta** (*torch.Tensor*) – The θ-component (polar angle) of spherical coordinates.
- **phi** (*torch.Tensor*) – The ϕ-component (azimuthal angle) of spherical coordinates.

Returns The x-, y-, and z-component in cartesian coordinates.

Return type tuple[*torch.Tensor*]

neurodiffeq.operators.**spherical_vector_laplacian**(*u_r*, *u_theta*, *u_phi*, *r*, *theta*, *phi*)

Derives and evaluates the spherical laplacian of a spherical vector field u.

Parameters

- **u_r** (*torch.Tensor*) – The r-component of the vector field u, must have shape (n_samples, 1).
- **u_theta** (*torch.Tensor*) – The θ-component of the vector field u, must have shape (n_samples, 1).
- **u_phi** (*torch.Tensor*) – The ϕ-component of the vector field u, must have shape (n_samples, 1).
- **r** (*torch.Tensor*) – A vector of r-coordinate values, must have shape (n_samples, 1).
- **theta** (*torch.Tensor*) – A vector of θ-coordinate values, must have shape (n_samples, 1).
- **phi** (*torch.Tensor*) – A vector of ϕ-coordinate values, must have shape (n_samples, 1).

Returns The laplacian evaluated at (r, θ, ϕ), with shape (n_samples, 1).

Return type *torch.Tensor*

neurodiffeq.operators.**vector_laplacian**(*u_x*, *u_y*, *u_z*, *x*, *y*, *z*)

Derives and evaluates the vector laplacian of a vector field $\mathbf{u}$ in three dimensional cartesian coordinates.

Parameters

- **u_x** (*torch.Tensor*) – The x-component of the vector field u, must have shape (n_samples, 1).
- **u_y** (*torch.Tensor*) – The y-component of the vector field u, must have shape (n_samples, 1).
- **u_z** (*torch.Tensor*) – The z-component of the vector field u, must have shape (n_samples, 1).
- **x** (*torch.Tensor*) – A vector of x-coordinate values, must have shape (n_samples, 1).
- **y** (*torch.Tensor*) – A vector of y-coordinate values, must have shape (n_samples, 1).
- **z** (*torch.Tensor*) – A vector of z-coordinate values, must have shape (n_samples, 1).

Returns Components of vector laplacian of $\mathbf{u}$ evaluated at $\mathbf{x}$, each with shape (n_samples, 1).

Return type tuple[*torch.Tensor*]

4.13 *neurodiffeq.callbacks*

class neurodiffeq.callbacks.**ActionCallback**(*logger=None*)

Bases: *neurodiffeq.callbacks.BaseCallback*

Base class of action callbacks. Custom callbacks that *performs an action* should subclass this class.

Parameters **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**AndCallback**(*condition_callbacks*, *logger=None*)

Bases: *neurodiffeq.callbacks.ConditionCallback*

A ConditionCallback which evaluates to True iff none of its sub-ConditionCallback s evaluates to False.

Parameters

- **condition_callbacks** (list[ConditionCallback]) – List of sub-ConditionCallback s.
- **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

Note: `c = AndCallback([c1, c2, c3])` can be simplified as `c = c1 & c2 & c3`.

---

class neurodiffeq.callbacks.**BaseCallback**(*logger=None*)

Bases: abc.ABC, neurodiffeq.callbacks._LoggerMixin

Base class of all callbacks. The class should not be directly subclassed. Instead, subclass *ActionCallback* or *ConditionCallback*.

Parameters **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**CheckpointCallback**(*ckpt_dir*, *logger=None*)
Bases: *neurodiffeq.callbacks.ActionCallback*

A callback that saves the networks (and their weights) to the disk.

Parameters

- **ckpt_dir** (*str*) – The directory to save model checkpoints. If non-existent, the directory is automatically created at instantiation time.
- **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

Note: Unless the callback is called twice within the same second, new checkpoints will not overwrite existing ones.

---

class neurodiffeq.callbacks.**ClosedIntervalGlobal**(*min=None*, *max=None*, *logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A ConditionCallback which evaluates to True only when $g_0 \leq g \leq g_1$, where g is the global epoch count.

Parameters

- **min** (*int*) – Lower bound of the closed interval (g_0 in the above inequality). Defaults to None.
- **max** (*int*) – Upper bound of the closed interval (g_1 in the above inequality). Defaults to None.
- **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**ClosedIntervalLocal**(*min=None*, *max=None*, *logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A ConditionCallback which evaluates to True only when $l_0 \leq l \leq l_1$, where l is the local epoch count.

Parameters

- **min** (*int*) – Lower bound of the closed interval (l_0 in the above inequality). Defaults to None.
- **max** (*int*) – Upper bound of the closed interval (l_1 in the above inequality). Defaults to None.
- **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**ConditionCallback**(*logger=None*)
Bases: *neurodiffeq.callbacks.BaseCallback*

Base class of condition callbacks. Custom callbacks that *determines whether some action shall be performed* should subclass this class and overwrite the .condition method.

Instances of ConditionCallback (and its children classes) support (short-circuit) evaluation of common boolean operations: & (and), | (or), ~ (not), and ^ (xor).

Parameters **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**EveCallback**(*base_value=1.0*, *double_at=0.1*, *n_0=1*, *n_max=None*, *use_train=True*, *metric='loss'*, *logger=None*)

Bases: *neurodiffeq.callbacks.ActionCallback*

A callback that readjusts the number of batches for training based on latest value of a specified metric. The number of batches will be $\left(n_0 \cdot 2^k\right)$ or $n_{\max}$ (if specified), whichever is lower, where $k = \max\left(0, \left\lfloor \log_p \frac{v}{v_0} \right\rfloor\right)$ and v is the value of the metric in the last epoch.

Parameters

- **base_value** (*float*) – Base value of the specified metric (v_0 in the above equation). When the metric value is higher than `base_value`, number of batches will be n_0.
- **double_at** (*float*) – The ratio at which the batch number will be doubled (p in the above equation). When $\frac{v}{v_0} = p^k$, the number of batches will be $\left(n_0 \cdot 2^k\right)$.
- **n_0** (*int*) – Minimum number of batches (n_0). Defaults to 1.
- **n_max** (*int*) – Maximum number of batches ($n_{\max}$). Defaults to infinity.
- **use_train** (*bool*) – Whether to use the training (instead of validation) phase value of the metric. Defaults to True.
- **metric** (*str*) – Name of which metric to use. Must be 'loss' or present in `solver.metrics_fn.keys()`. Defaults to 'loss'.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**FalseCallback**(*logger=None*)

Bases: *neurodiffeq.callbacks.ConditionCallback*

A `ConditionCallback` which always evaluates to False.

Parameters **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**MonitorCallback**(*monitor*, *fig_dir=None*, *format=None*, *logger=None*, ***kwargs*)

Bases: *neurodiffeq.callbacks.ActionCallback*

A callback for updating the monitor plots (and optionally saving the fig to disk).

Parameters

- **monitor** (*neurodiffeq.monitors.BaseMonitor*) – The underlying monitor responsible for plotting solutions.
- **fig_dir** (*str*) – Directory for saving monitor figs; if not specified, figs will not be saved.
- **format** (*str*) – Format for saving figures: {'jpg', 'png' (default), ...}.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**NotCallback**(*condition_callback*, *logger=None*)

Bases: *neurodiffeq.callbacks.ConditionCallback*

A `ConditionCallback` which evaluates to True iff its sub-`ConditionCallback` evaluates to False.

Parameters

- **condition_callback** (ConditionCallback) – The sub-ConditionCallback .
- **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

Note: `c = NotCallback(c1)` can be simplified as `c = ~c1`.

class neurodiffeq.callbacks.**OnFirstGlobal**(*logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A ConditionCallback which evaluates to True only on the first global epoch.

Parameters **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**OnFirstLocal**(*logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A ConditionCallback which evaluates to True only on the first local epoch.

Parameters **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**OnLastLocal**(*logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A ConditionCallback which evaluates to True only on the last local epoch.

Parameters **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**OrCallback**(*condition_callbacks*, *logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A ConditionCallback which evaluates to False iff none of its sub-ConditionCallback s evaluates to True.

Parameters

- **condition_callbacks** (list[ConditionCallback]) – List of sub-ConditionCallback s.
- **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

Note: `c = OrCallback([c1, c2, c3])` can be simplified as `c = c1 | c2 | c3`.

class neurodiffeq.callbacks.**PeriodGlobal**(*period*, *offset=0*, *logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A ConditionCallback which evaluates to True only when the global epoch count equals $\text{period}\times n+\text{offset}$.

Parameters

- **period** (*int*) – Period of the callback.
- **offset** (*int*) – Offset of the period. Defaults to 0.
- **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**PeriodLocal**(*period*, *offset=0*, *logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A `ConditionCallback` which evaluates to True only when the local epoch count equals $\text{period} \times n + \text{offset}$.

Parameters

- **period** (*int*) – Period of the callback.
- **offset** (*int*) – Offset of the period. Defaults to 0.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**ProgressBarCallBack**(*logger=None*)
Bases: *neurodiffeq.callbacks.ActionCallback*

class neurodiffeq.callbacks.**Random**(*probability*, *logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A `ConditionCallback` which has a certain probability of evaluating to True.

Parameters

- **probability** (*float*) – The probability of this callback evaluating to True (between 0 and 1).
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**RepeatedMetricAbove**(*threshold*, *use_train*, *metric*, *repetition*, *logger*)
Bases: `neurodiffeq.callbacks._RepeatedMetricChange`

A `ConditionCallback` which evaluates to True if a certain metric has been greater than a given value v for the latest n epochs.

Parameters

- **threshold** (*float*) – The value *v*.
- **use_train** (*bool*) – Whether to use the metric value in the training (rather than validation) phase.
- **metric** (*str*) – Name of which metric to use. Must be 'loss' or present in `solver.metrics_fn.keys()`. Defaults to 'loss'.
- **repetition** (*int*) – Number of times the metric should diverge beyond said gap.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**RepeatedMetricBelow**(*threshold*, *use_train*, *metric*, *repetition*, *logger*)
Bases: `neurodiffeq.callbacks._RepeatedMetricChange`

A `ConditionCallback` which evaluates to True if a certain metric has been less than a given value v for the latest n epochs.

Parameters

- **threshold** (*float*) – The value *v*.
- **use_train** (*bool*) – Whether to use the metric value in the training (rather than validation) phase.

- **metric** (*str*) – Name of which metric to use. Must be 'loss' or present in `solver.metrics_fn.keys()`. Defaults to 'loss'.
- **repetition** (*int*) – Number of times the metric should diverge beyond said gap.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**RepeatedMetricConverge**(*epsilon*, *use_train=True*, *metric='loss'*, *repetition=1*, *logger=None*)

Bases: `neurodiffeq.callbacks._RepeatedMetricChange`

A `ConditionCallback` which evaluates to True if a certain metric for the latest n epochs kept converging within some tolerance ε.

Parameters

- **epsilon** (*float*) – The said tolerance.
- **use_train** (*bool*) – Whether to use the metric value in the training (rather than validation) phase.
- **metric** (*str*) – Name of which metric to use. Must be 'loss' or present in `solver.metrics_fn.keys()`. Defaults to 'loss'.
- **repetition** (*int*) – Number of times the metric should converge within said tolerance.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**RepeatedMetricDiverge**(*gap*, *use_train=True*, *metric='loss'*, *repetition=1*, *logger=None*)

Bases: `neurodiffeq.callbacks._RepeatedMetricChange`

A `ConditionCallback` which evaluates to True if a certain metric for the latest n epochs kept diverging beyond some gap.

Parameters

- **gap** (*float*) – The said gap.
- **use_train** (*bool*) – Whether to use the metric value in the training (rather than validation) phase.
- **metric** (*str*) – Name of which metric to use. Must be 'loss' or present in `solver.metrics_fn.keys()`. Defaults to 'loss'.
- **repetition** (*int*) – Number of times the metric should diverge beyond said gap.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**RepeatedMetricDown**(*at_least_by=0.0*, *use_train=True*, *metric='loss'*, *repetition=1*, *logger=None*)

Bases: `neurodiffeq.callbacks._RepeatedMetricChange`

A `ConditionCallback` which evaluates to True if a certain metric for the latest n epochs kept decreasing by at least some margin.

Parameters

- **at_least_by** (*float*) – The said margin.
- **use_train** (*bool*) – Whether to use the metric value in the training (rather than validation) phase.

- **metric** (*str*) – Name of which metric to use. Must be 'loss' or present in `solver.metrics_fn.keys()`. Defaults to 'loss'.
- **repetition** (*int*) – Number of times the metric should decrease by the said margin (the n).
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**RepeatedMetricUp**(*at_least_by=0.0*, *use_train=True*, *metric='loss'*, *repetition=1*, *logger=None*)

Bases: `neurodiffeq.callbacks._RepeatedMetricChange`

A `ConditionCallback` which evaluates to True if a certain metric for the latest n epochs kept increasing by at least some margin.

Parameters

- **at_least_by** (*float*) – The said margin.
- **use_train** (*bool*) – Whether to use the metric value in the training (rather than validation) phase.
- **metric** (*str*) – Name of which metric to use. Must be 'loss' or present in `solver.metrics_fn.keys()`. Defaults to 'loss'.
- **repetition** (*int*) – Number of times the metric should increase by the said margin (the n).
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**ReportCallback**(*logger=None*)

Bases: *neurodiffeq.callbacks.ActionCallback*

A callback that logs the training/validation information, including

- number of batches (train/valid)
- batch size (train/valid)
- generator to be used (train/valid)

Parameters **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

neurodiffeq.callbacks.**ReportOnFitCallback**(*logger=None*)

A callback that logs the training/validation information, including

- number of batches (train/valid)
- batch size (train/valid)
- generator to be used (train/valid)

Parameters **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

neurodiffeq.callbacks.**SetCriterion**(*loss_fn*, *reset=False*, *logger=None*)

A callback that sets the `criterion` (a.k.a. loss function) of the solver. Best used together with a condition callback.

Parameters

- **loss_fn** (torch.nn.modules.loss._Loss or callable or str.) – The loss function to be set for the solver. It can be
  - An instance of torch.nn.modules.loss._Loss which computes loss of the PDE/ODE residuals against a zero tensor.
  - A callable object which maps residuals, function values, and input coordinates to a scalar loss; or
  - A str which is present in neurodiffeq.losses._losses.keys().
- **reset** (*bool*) – If True, the criterion will be reset every time the callback is called. Otherwise, the criterion will only be set once. Defaults to False.
- **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**SetLossFn**(*loss_fn*, *reset=False*, *logger=None*)

Bases: *neurodiffeq.callbacks.ActionCallback*

A callback that sets the criterion (a.k.a. loss function) of the solver. Best used together with a condition callback.

Parameters

- **loss_fn** (torch.nn.modules.loss._Loss or callable or str.) – The loss function to be set for the solver. It can be
  - An instance of torch.nn.modules.loss._Loss which computes loss of the PDE/ODE residuals against a zero tensor.
  - A callable object which maps residuals, function values, and input coordinates to a scalar loss; or
  - A str which is present in neurodiffeq.losses._losses.keys().
- **reset** (*bool*) – If True, the criterion will be reset every time the callback is called. Otherwise, the criterion will only be set once. Defaults to False.
- **logger** (str or logging.Logger) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**SetOptimizer**(*optimizer*, *optimizer_args=None*, *optimizer_kwargs=None*, *reset=False*, *logger=None*)

Bases: *neurodiffeq.callbacks.ActionCallback*

A callback that sets the optimizer of the solver. Best used together with a condition callback.

- If an optimizer *instance* is passed, it must contain a sequence of parameters to be updated.
- If an optimizer *subclass* is passed, optimizer_args and optimizer_kwargs can be supplied.

Parameters

- **optimizer** (type or torch.optim.Optimizer) – Optimizer instance (or its class) to be set.
- **optimizer_args** (*tuple*) – Positional arguments to be passed to the optimizer constructor in addition to the parameter sequence. Ignored if optimizer is an instance (instead of a class).
- **optimizer_kwargs** (*dict*) – Keyword arguments to be passed to the optimizer constructor in addition to the parameter sequence. Ignored if optimizer is an instance (instead of a class).

- **reset** (*bool*) – If True, the optimizer will be reset every time the callback is called. Otherwise, the optimizer will only be set once. Defaults to False.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**SimpleTensorboardCallback**(*writer=None*, *logger=None*)
Bases: *neurodiffeq.callbacks.ActionCallback*

A callback that writes all metric values to the disk for TensorBoard to plot. Tensorboard must be installed for this callback to work.

Parameters

- **writer** (`torch.utils.tensorboard.SummaryWriter`) – The summary writer for writing values to disk. Defaults to a new `SummaryWriter` instance created with default kwargs.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**StopCallback**(*logger=None*)
Bases: *neurodiffeq.callbacks.ActionCallback*

A callback that stops the training/validation process and terminates the `solver.fit()` call.

Parameters **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

Note: This callback should always be used together with a *ConditionCallback*, otherwise the `solver.fit()` call will exit after first epoch.

---

class neurodiffeq.callbacks.**TrueCallback**(*logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A `ConditionCallback` which always evaluates to True.

Parameters **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

class neurodiffeq.callbacks.**XorCallback**(*condition_callbacks*, *logger=None*)
Bases: *neurodiffeq.callbacks.ConditionCallback*

A `ConditionCallback` which evaluates to False iff evenly many of its sub-`ConditionCallback` s evaluates to True.

Parameters

- **condition_callbacks** (list[`ConditionCallback`]) – List of sub-`ConditionCallback` s.
- **logger** (str or `logging.Logger`) – The logger (or its name) to be used for this callback. Defaults to the 'root' logger.

Note: `c = XorCallback([c1, c2, c3])` can be simplified as `c = c1 ^ c2 ^ c3`.

4.14 *neurodiffeq.utils*

neurodiffeq.utils.**set_seed**(*seed_value*, *ignore_numpy=False*, *ignore_torch=False*, *ignore_random=False*)

Set the random seed for the numpy, torch, and random packages.

Parameters

- **seed_value** (*int*) – The value of seed.
- **ignore_numpy** (*bool*) – If True, the seed for numpy.random will not be set. Defaults to False.
- **ignore_torch** (*bool*) – If True, the seed for torch will not be set. Defaults to False.
- **ignore_random** (*bool*) – If True, the seed for *random* will not be set. Defaults to False.

neurodiffeq.utils.**set_tensor_type**(*device=None*, *float_bits=32*)

Set the default torch tensor type to be used with neurodiffeq.

Parameters

- **device** (*str*) – Either "cpu" or "cuda" ("gpu"); defaults to "cuda" if available.
- **float_bits** (*int*) – Length of float numbers. Either 32 (float) or 64 (double); defaults to 32.

CHAPTER 5

How Does It Work?

To solve a differential equation. We need the solution to satisfy 2 things: 1. They need to satisfy the equation 2. They need to satisfy the initial/boundary conditions

5.1 Satisfying the Equation

The key idea of solving differential equations with ANNs is to reformulate the problem as an optimization problem in which we minimize the residual of the differential equations. In a very general sense, a differential equation can be expressed as

$$\mathcal{L}u - f = 0$$

where $\mathcal{L}$ is the differential operator, $u(x,t)$ is the solution that we wish to find, and f is a known forcing function. Note that $\mathcal{L}$ contains any combination of temporal and spatial partial derivatives. Moreover, x is in general a vector in three spatial dimensions and u is a vector of solutions. We denote the output of the neural network as $u_N(x,t;p)$ where the parameter vector p is a vector containing the weights and biases of the neural network. We will drop the arguments of the neural network solution in order to be concise. If u_N is a solution to the differential equation, then the residual

$$\mathcal{R}(u_N) = \mathcal{L}u_N - f$$

will be identically zero. One way to incorporate this into the training process of a neural network is to use the residual as the loss function. In general, the L^2 loss of the residual is used. This is the convention that `NeuroDiffEq` follows, although we note that other loss functions could be conceived. Solving the differential equation is re-cast as the following optimization problem:

$$\min_p \left(\mathcal{L}u_N - f\right)^2.$$

5.2 Satisfying the Initial/Boundary Conditions

It is necessary to inform the neural network about any boundary and initial conditions since it has no way of enforcing these *a priori*. There are two primary ways to satisfy the boundary and initial conditions. First, one can impose the

initial/boundary conditions in the loss function. For example, given an initial condition $u(x, t_0) = u_0(x)$, the loss function can be modified to:

$$\min_p \left[(\mathcal{L}u_N - f)^2 + \lambda \left(u_N(x, t_0) - u_0(x) \right)^2 \right]$$

where the second term penalizes solutions that don't satisfy the initial condition. Larger values of the regularization parameter λ result in stricter satisfaction of the initial condition while sacrificing solution accuracy. However, this approach does not lead to *exact* satisfaction of the initial and boundary conditions.

Another option is to transform the u_N in a way such that the initial/boundary conditions are satisfied by construction. Given an initial condition $u_0(x)$ the neural network can be transformed according to:

$$\widetilde{u}(x, t) = u_0(x) + \left(1 - e^{-(t - t_0)} \right) u_N(x, t)$$

so that when $t = t_0$, $\widetilde{u}$ will always be u_0. Accordingly, the objective function becomes

$$\min_p \left(\mathcal{L}\widetilde{u} - f \right)^2.$$

This approach is similar to the trial function approach, but with a different form of the trial function. Modifying the neural network to account for boundary conditions can also be done. In general, the transformed solution will have the form:

$$\widetilde{u}(x, t) = A(x, t; x_{\text{boundary}}, t_0)\, u_N(x, t)$$

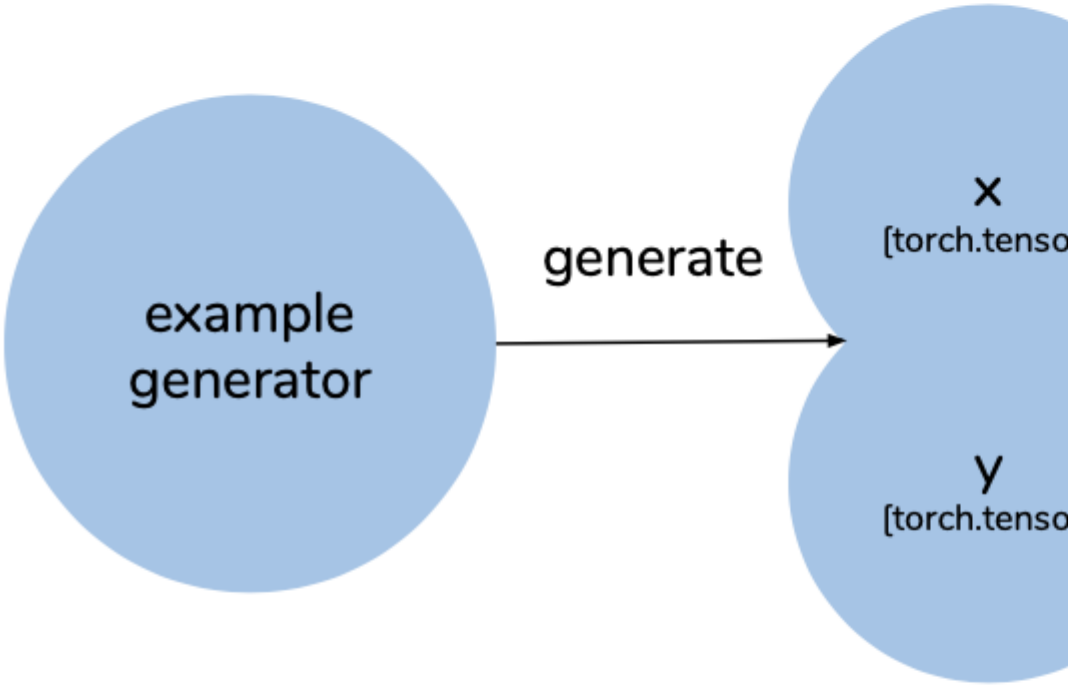
where $A(x, t; x_{\text{boundary}}, t_0)$ must be designed so that $\widetilde{u}(x, t)$ has the correct boundary conditions. This can be very challenging for complicated domains.

Both of these two methods have their advantages. The first way is simpler to implement and can be more easily extended to high-dimensional PDEs and PDEs formulated on complicated domains. The second way assures that the initial/boundary conditions are exactly satisfied. Considering that differential equations can be sensitive to initial/boundary conditions, this is expected to play an important role. Another advantage of the second method is that fixing these conditions can reduce the effort required during the training of the ANN. `NeuroDiffEq` employs the second approach.

5.3 The implementation

We minimize the loss function with optimization algorithms implemented in `torch.optim.optimizer`. These algorithms use the derivatives of the loss function w.r.t. the network weights to update the network weights. The derivatives are computed with `torch.autograd`, which implements reverse mode automatic differentiation. Automatic differentiation calculates the derivatives base on the computational graph from the network weights to the loss, so discretization is not required. Here, our loss function is a bit different from the ones usually found in deep learning in the sense that, it not only involves the network output but also the derivatives of the network output w.r.t. the input. This second kind of derivatives is computed with `torch.autograd` as well.

Most of the logic of the above-mentioned method is implemented in `ode.solve` and `pde.solve2D` functions. The following diagram shows the process flow of solving a PDE $\mathcal{L}u - f = 0$ for $u(x, y)$. The PDE is passed as a function that takes in x, y and u and produces the value of $\mathcal{L}u - f$. To specify the differential operator $\mathcal{L}$, the user can make use of the function `neurodiffeq.diff` which is just a wrapper of `torch.autograd.grad`. During each training epoch, a set of x and y is generated as `torch.tensor`. They are split into multiple batches. Each batch is used to make a little tweak to the network weights. In each iteration, a batch of x, y are fed into the neural network, the output then transformed to impose the boundary conditions. This gives us the approximate solution $\tilde{u}$. x, y and $\tilde{u}$ are then passed to the PDE function specified by user to calculate $\mathcal{L}\tilde{u} - f$ and the final loss function (the default is just $(\mathcal{L}\tilde{u} - f)^2$). The optimizer then adjusts the network weights to better satisfy the PDE by taking a step to minimize the loss function. We repeat these steps until we reach the maximum number of epochs specified by the user.


example
generator
generate
x
[torch.tenso
y
[torch.tenso
neu
(im
t

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

X

Z