torchphysics Documentation

Release unknown

nick7

Jun 28, 2022

CONTENTS

Welcome to **TorchPhysics**, a Python library of deep learning methods for solving differential equations. Currently, TorchPhysics implements methods like PINN[1] and DeepRitz[2] which enable the user to

- solve ordinary and partial differential equations
- train a neural network to approximate solutions for different parameters
- solve inverse problems and interpolate external data via the above methods

TorchPhysics can also be used in other deep learning approaches for differential equations since it is built in a modular way. For example, TorchPhysics offers a way to sample points in arbitrary, easy-to-define, domains flexibly.

[1] Raissi, Perdikaris und Karniadakis, "Physics-informed neuralnetworks: A deep learning framework for solving forward and inverseproblems involving nonlinear partial differential equations", 2019.

[2] E and Yu, "The Deep Ritz method: A deep learning-based numerical algorithm for solving variational problems", 2017

CHAPTER

ONE

GUIDE

All kind of information (features, installation, etc.) to TorchPhysics can be found under the **Overview** tab.

As an introduction to TorchPhysics a **Tutorial** exists. There we will present and explain the most important aspects and structure of this library. Under the **Examples** tab additional applications, in form of Jupyter Notebooks, can be found.

1.1 TorchPhysics

TorchPhysics is a Python library of (mesh-free) deep learning methods to solve differential equations. You can use TorchPhysics e.g. to

- solve ordinary and partial differential equations
- train a neural network to approximate solutions for different parameters
- solve inverse problems and interpolate external data

The following approaches are implemented using high-level concepts to make their usage as easy as possible:

- physics-informed neural networks (PINN)[1]
- QRes[2]
- the Deep Ritz method[3]

TorchPhysics can also be used to implement extensions of these approaches or concepts like DeepONets[4] and Physics-Informed DeepONets[5]. We aim to also include further implementations in the future.

TorchPhysics is build upon the machine learning library PyTorch.

[1] Raissi, Perdikaris und Karniadakis, "Physics-informed neuralnetworks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations", 2019.

[2] Bu and Karpatne, "Quadratic Residual Networks: A New Class of Neural Networks for Solving Forward and Inverse Problems in Physics Involving PDEs", 2021

[3] E and Yu, "The Deep Ritz method: A deep learning-based numerical algorithm for solving variational problems", 2017

[4] Lu, Jin and Karniadakis, "DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators", 2020

[5] Wang, Wang and Perdikaris, "Learning the solution operator of parametric partial differential equations with physics-informed DeepOnets", 2021

1.1.1 Features

The Goal of this library is to create a basic framework that can be used in many different applications and with different deep learning methods. To this end, TorchPhysics aims at a:

- modular and expandable structure
- easy to understand code and clean documentation
- intuitive and compact way to transfer the mathematical problem into code
- reliable and well tested code basis

Some built-in features are:

- mesh free domain generation. With pre implemented domain types: *Point, Interval, Parallelogram, Circle, Triangle and Sphere*
- loading external created objects, thanks to a soft dependency on Trimesh and Shapely
- creating complex domains with the boolean operators *Union*, *Cut* and *Intersection* and higher dimensional objects over the Cartesian product
- allowing interdependence of different domains, e.g. creating moving domains
- different point sampling methods for every domain: *RandomUniform, Grid, Gaussian, Latin hypercube, Adaptive* and some more for specific domains
- different operators to easily define a differential equation
- pre implemented fully connected neural network and easy implementation of additional model structures
- sequentially or parallel evaluation/training of different neural networks
- normalization layers and adaptive weights[6] to speed up the training process
- powerful and versatile training thanks to PyTorch Lightning
  - many options for optimizers and learning rate control
  - monitoring the loss of individual conditions while training

1.1.2 Getting Started

To learn the functionality and usage of TorchPhysics we recommend to have a look at the following sections:

- Tutorial: Understanding the structure of TorchPhysics
- Examples: Different applications with detailed explanations
- Documentation

[6] McClenny und Braga-Neto, "Self-Adaptive Physics-Informed NeuralNetworks using a Soft Attention Mechanism", 2020

1.1.3 Installation

TorchPhysics can be installed by using:

```
pip install git+https://github.com/boschresearch/torchphysics
```

If you want to change or add something to the code. You should first copy the repository and install it locally:

```
git clone https://github.com/boschresearch/torchphysics
pip install .
```

1.1.4 About

TorchPhysics was originally developed by Nick Heilenkötter and Tom Freudenberg, as part of a seminar project at the University of Bremen, in cooperation with the Robert Bosch GmbH. Special thanks belong to Felix Hildebrand, Uwe Iben, Daniel Christopher Kreuter and Johannes Mueller, at the Robert Bosch GmbH, for support and supervision while creating this library.

1.1.5 Contribute

If you are missing a feature or detect a bug or unexpected behaviour while using this library, feel free to open an issue or a pull request in GitHub or contact the authors. Since we developed the code as a student project during a seminar, we cannot guarantee every feature to work properly. However, we are happy about all contributions since we aim to develop a reliable code basis and extend the library to include other approaches.

1.1.6 License

TorchPhysics uses an Apache License, see the LICENSE file.

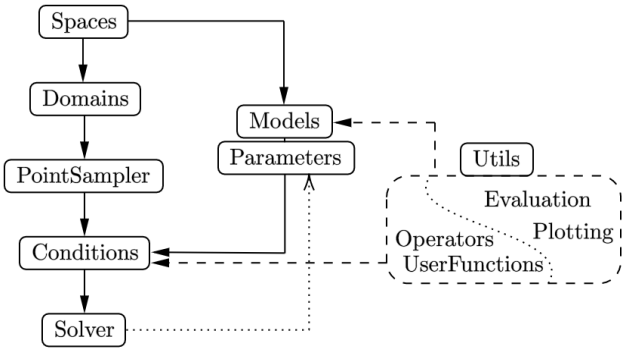
1.1.7 Bibliography

1.2 Tutorial: Understanding the structure of TorchPhysics

In this tutorial, you will learn how the different components of TorchPhysics work and interact. In the end, you will be able to transform a simple differential equation into the corresponding training setup in TorchPhysics.

We start by explaining the basic structure of this library. Then we will go over each important part in its own tutorial. Finally we will bring everything together to solve a PDE using the PINN-approach.

The structure of TorchPhysics can be illustrated via the following graph:



Spaces
Define the dimension of the used variables, parameters and model outputs. This is the starting point for all problems and shown in the spaces and points tutorial. The tutorial also covers the way the library stores and creates points.

Domains
Handle the creation of the underlying geometry of the problem. See:

- Domain basics to learn everything about the definition and functionalities
- Polygons and external objects to create 2D or 3D polygons and import external files

PointSampler
Control the creation of sampling points for the training/validation process. The usage is explained in the PointSampler tutorial.

Models/Parameters
Implement different neural network structures and trainable parameters. How to define a model is shown in the model creation tutorial.

Conditions
Combine the created *Domains*, *PointSampler* and *Models* to apply the conditions induced by the differential equation. See condition tutorial on how to create different kinds of conditions for all parts of the problem.

Utils
Implement a variety of helper functions to make the definition and evaluation of problems easier. To get an overview of all methods, see the docs. Two parts that will be shown more detailed, are:

- The usage of the pre implemented differential operators
- Creating plots of the trained solutions.

Solver
Handles the training of the defined model, by applying the previously created conditions. The usage of the solver will be shown in a complete problem, where all the above parts of the library are used. This is shown in solving a simple PDE.

These are all the basics of TorchPhysics. You should now have a rough understanding of the structure of this library. Some additional applications (inverse problems, training input params, ...) can be found under the example-folder

1.3 Examples

Here we want to mention and link some examples and applications that, in our opinion, nicely present the functionalities of TorchPhysics and the PINN idea.

More examples can be found under the examples-folder.

1.3.1 Poisson problem

One of the simplest applications is the forward solution of a Poisson equation:

$$\begin{aligned} \Delta u &= 4.25\pi^2 u, \text{ in } \Omega = [0,1] \times [0,1] \\ u &= \sin(\frac{\pi}{2} x_1)\cos(2\pi x_2), \text{ on } \partial\Omega \end{aligned} \tag{1.1}$$

This problem is part of the tutorial and only mentioned for completeness. The corresponding implementation can be found here.

1.3.2 Learning parameter dependencies

A natural extension of the PINN approach is to learn parameter dependencies, that appear in the differential equation. A simple example would be the problem:

$$\begin{aligned} \partial_x u &= ku, \text{ in } [0,1] \\ u(0) &= 1 \end{aligned}$$

where we want to train a family of solutions for $k \in [0,2]$. So we essentially want to find the function $u(x,k) = e^{kx}$. Implemented is this example in: simple-parameter-dependency-notebook

This approach is also possible for complexer problems, see for example this notebook. Where we apply this idea to the heat equation.

1.3.3 Inverse heat equation

For an inverse problem we consider the heat equation:

$$\begin{aligned} \text{div}(D(x)\nabla u(x,t)) &= \partial_t u(x,t), \text{ in } \Omega \times [0,5] \\ u(x,0) &= 100\sin(\tfrac{\pi}{10}x_1)\sin(\tfrac{\pi}{10}x_2), \text{ in } \Omega \\ u(x,t) &= 0, \text{ on } \partial\Omega \times [0,5] \end{aligned}$$

with $\Omega = [0,10] \times [0,10]$. Here D can either be a constant value or function itself. Here we start with some data $u(t_i, x_i)$ and want to find the corresponding D.

The aim of the following two examples is to show how one can implement this in TorchPhysics:

- Constant-D-notebook
- Space-dependent-D-notebook

1.3.4 Heat equation on moving domain

To demonstrate how easy one can create a time (or parameter) dependent domain, we consider the PDE:

$$\begin{aligned} \partial_t u - D\Delta u &= 0, \text{ in } \Omega \times [0, T] \\ u(\cdot, 0) &= 0, \text{ in } \Omega \\ u &= 0, \text{ on } \Gamma_{\text{out}} \times [0, T] \\ \vec{n}\nabla u &= q_{\text{in}}, \text{ on } \Gamma_{\text{in}}(t) \times [0, T] \end{aligned}$$

Where Ω will be a circle with a moving hole and $\Gamma_{\text{in}}(t)$ the boundary of the hole. The animation on the main page belongs to the solution of this problem.

Link to the notebook: moving-domain-notebook

1.3.5 Interface jump

For an example where we want to solve a problem with a discontinuous solution, we study, for $\Omega = (0, 1) \times (0, 1)$ and Γ the line form $(0.5, 0)$ to $(0.5, 1)$, the PDE:

$$\begin{aligned} \Delta u_i &= 0, \text{ in } \Omega \\ u_1(0, y) &= 0, \text{ for } y \in [0, 1] \\ u_2(1, y) &= 2, \text{ for } y \in [0, 1] \\ \vec{n}\nabla u_i(x, y) &= 0, \text{ for } x \in [0, 1], y \in \{0, 1\} \\ \vec{n}\nabla u_i &= u_2 - u_1, \text{ for } x \in \Gamma \end{aligned}$$

with $i = 1, 2$ and the solution $u = (u_1, u_2)$, split up into left and right part.

For this problem we need two networks, since one alone can, in general, not approximate the jump of the solution. Therefore, this example focus on the training of two neural networks on disjoint domains, coupled over the interface.

Link to the notebook: jump-notebook

CHAPTER

TWO

API REFERENCE

Information for all classes, functions and methods can be found in the following documentation:

2.1 torchphysics.problem.conditions package

Conditions are the central concept in this package. They supply the necessary training data to the model and translate the condition of the differential equation into the trainings condition of the neural network.

A tutorial on the usage of Conditions can be found here.

2.1.1 Submodules

2.1.2 torchphysics.problem.conditions.condition module

class torchphysics.problem.conditions.condition.**AdaptiveWeightsCondition**(*module*, *sampler*, *residual_fn*, *error_fn=SquaredError()*, *track_gradients=True*, *data_functions={}*, *parameter=Parameter: {}*, *name='adaptive_w_condition'*, *weight=1.0*)

Bases: *SingleModuleCondition*

A condition using an AdaptiveWeightLayer [1] to assign adaptive weights to all points during training.

Parameters

- **module** (*torchphysics.Model*) – The torch module which should be optimized.
- **sampler** (*torchphysics.samplers.PointSampler*) – A sampler that creates the points in the domain of the residual function, could be an inner or a boundary domain.
- **residual_fn** (*callable*) – A user-defined function that computes the residual (unreduced loss) from inputs and outputs of the model, e.g. by using utils.differentialoperators and/or domain.normal
- **error_fn** (*callable*) – Function that will be applied to the output of the residual_fn to compute the unreduced loss (shape [n_points]). The result will be multiplied by the adaptive weights.

- **data_functions** (*dict*) – A dictionary of user-defined functions and their names (as keys). Can be used e.g. for right sides in PDEs or functions in boundary conditions.
- **track_gradients** (*bool*) – Whether gradients w.r.t. the inputs should be tracked during training or not. Defaults to true, since this is needed to compute differential operators in PINNs.
- **parameter** (Parameter) – A Parameter that can be used in the residual_fn and should be learned in parallel, e.g. based on data (in an additional DataCondition).
- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.

Notes

training: bool

class torchphysics.problem.conditions.condition.**Condition**(*name=None*, *weight=1.0*, *track_gradients=True*)

Bases: Module

A general condition which should be optimized or tracked.

Parameters

- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.
- **track_gradients** (*bool*) – Whether to track input gradients or not. Helps to avoid tracking the gradients during validation. If a condition is applied during training, the gradients will always be tracked.

abstract forward(*device='cpu'*, *iteration=None*)

The forward run performed by this condition.

Returns
torch.Tensor

Return type
the loss which should be minimized or monitored during training

training: bool

class torchphysics.problem.conditions.condition.**DataCondition**(*module*, *dataloader*, *norm*, *use_full_dataset=False*, *name='datacondition'*, *weight=1.0*)

Bases: *Condition*

A condition that fits a single given module to data (handed through a PyTorch dataloader).

Parameters

- **module** (*torchphysics.Model*) – The torch module which should be fitted to data.
- **dataloader** (*torch.utils.DataLoader*) – A PyTorch dataloader which supplies the iterator to load data-target pairs from some given dataset. Data and target should be handed as points in input or output spaces, i.e. with the correct point object.

- **norm** (*int or 'inf'*) – The 'norm' which should be computed for evaluation. If 'inf', maximum norm will be used. Else, the result will be taken to the n-th potency (without computing the root!)
- **use_full_dataset** (*bool*) – Whether to perform single iterations or compute the error on the whole dataset during forward call. The latter can especially be useful during validation.
- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.

forward(*device='cpu'*, *iteration=None*)

The forward run performed by this condition.

Returns
torch.Tensor

Return type
the loss which should be minimized or monitored during training

training: bool

class torchphysics.problem.conditions.condition.**DeepRitzCondition**(*module*, *sampler*, *integrand_fn*, *track_gradients=True*, *data_functions={}*, *parameter=Parameter: {}*, *name='deepritzcondition'*, *weight=1.0*)

Bases: *MeanCondition*

Alias for *MeanCondition*.

Parameters

- **module** (*torchphysics.Model*) – The torch module which should be optimized.
- **sampler** (*torchphysics.samplers.PointSampler*) – A sampler that creates the points in the domain of the residual function, could be an inner or a boundary domain.
- **integrand_fn** (*callable*) – The integrand of the weak formulation of the differential equation.
- **data_functions** (*dict*) – A dictionary of user-defined functions and their names (as keys). Can be used e.g. for right sides in PDEs or functions in boundary conditions.
- **track_gradients** (*bool*) – Whether gradients w.r.t. the inputs should be tracked during training or not. Defaults to true, since this is needed to compute differential operators in PINNs.
- **parameter** (Parameter) – A Parameter that can be used in the residual_fn and should be learned in parallel, e.g. based on data (in an additional DataCondition).
- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.

Notes

training: bool

class torchphysics.problem.conditions.condition.**IntegroPINNCondition**(*module*, *sampler*, *residual_fn*, *integral_sampler*, *error_fn=SquaredError()*, *reduce_fn=<built-in method mean of type object>*, *name='periodiccondition'*, *track_gradients=True*, *data_functions={}*, *parameter=Parameter: {}*, *weight=1.0*)

Bases: *Condition*

A condition that also allows to include the computation of integrals or convolutions by sampling a second set of points by an additional sampler.

Parameters

- **module** (*torchphysics.Model*) – The torch module which should be optimized.
- **sampler** (*torchphysics.samplers.PointSampler*) – A sampler that creates the usual set of points.
- **integral_sampler** (*torchphysics.samplers.PointSampler*) – A sampler that creates the points that can be used to approximate an integral.
- **residual_fn** (*callable*) – A user-defined function that computes the residual (unreduced loss) from inputs and outputs of the model, e.g. by using utils.differentialoperators and/or domain.normal. The point set used to approximate the integral and the output of the model at these points are given as input {name}_integral
- **error_fn** (*callable*) – Function that will be applied to the output of the residual_fn to compute the unreduced loss. Should reduce only along the 2nd (i.e. space-)axis.
- **reduce_fn** (*callable*) – Function that will be applied to reduce the loss to a scalar. Defaults to torch.mean
- **data_functions** (*dict*) – A dictionary of user-defined functions and their names (as keys). Can be used e.g. for right sides in PDEs or functions in boundary conditions.
- **track_gradients** (*bool*) – Whether gradients w.r.t. the inputs should be tracked during training or not. Defaults to true, since this is needed to compute differential operators in PINNs.
- **parameter** (Parameter) – A Parameter that can be used in the residual_fn and should be learned in parallel, e.g. based on data (in an additional DataCondition).
- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.

forward(*device='cpu'*, *iteration=None*)

The forward run performed by this condition.

Returns

torch.Tensor

Return type
the loss which should be minimized or monitored during training

training: bool

class torchphysics.problem.conditions.condition.**MeanCondition**(*module*, *sampler*, *residual_fn*, *track_gradients=True*, *data_functions={}*, *parameter=Parameter: {}*, *name='meancondition'*, *weight=1.0*)

Bases: *SingleModuleCondition*

A condition that minimizes the mean of the residual of a single module, can be used e.g. in Deep Ritz Method [1] or for energy functionals, since the mean can be seen as a (scaled) integral approximation.

Parameters

- **module** (*torchphysics.Model*) – The torch module which should be optimized.
- **sampler** (*torchphysics.samplers.PointSampler*) – A sampler that creates the points in the domain of the residual function, could be an inner or a boundary domain.
- **residual_fn** (*callable*) – A user-defined function that computes the residual (unreduced loss) from inputs and outputs of the model, e.g. by using utils.differentialoperators and/or domain.normal
- **data_functions** (*dict*) – A dictionary of user-defined functions and their names (as keys). Can be used e.g. for right sides in PDEs or functions in boundary conditions.
- **track_gradients** (*bool*) – Whether gradients w.r.t. the inputs should be tracked during training or not. Defaults to true, since this is needed to compute differential operators in PINNs.
- **parameter** (Parameter) – A Parameter that can be used in the residual_fn and should be learned in parallel, e.g. based on data (in an additional DataCondition).
- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.

Notes

training: bool

class torchphysics.problem.conditions.condition.**PINNCondition**(*module*, *sampler*, *residual_fn*, *track_gradients=True*, *data_functions={}*, *parameter=Parameter: {}*, *name='pinncondition'*, *weight=1.0*)

Bases: *SingleModuleCondition*

A condition that minimizes the mean squared error of the given residual, as required in the framework of physics-informed neural networks [1].

Parameters

- **module** (*torchphysics.Model*) – The torch module which should be optimized.
- **sampler** (*torchphysics.samplers.PointSampler*) – A sampler that creates the points in the domain of the residual function, could be an inner or a boundary domain.

- **residual_fn** (*callable*) – A user-defined function that computes the residual (unreduced loss) from inputs and outputs of the model, e.g. by using utils.differentialoperators and/or domain.normal
- **data_functions** (*dict*) – A dictionary of user-defined functions and their names (as keys). Can be used e.g. for right sides in PDEs or functions in boundary conditions.
- **track_gradients** (*bool*) – Whether gradients w.r.t. the inputs should be tracked during training or not. Defaults to true, since this is needed to compute differential operators in PINNs.
- **parameter** (Parameter) – A Parameter that can be used in the residual_fn and should be learned in parallel, e.g. based on data (in an additional DataCondition).
- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.

Notes

training: bool

class torchphysics.problem.conditions.condition.**ParameterCondition**(*parameter*, *penalty*, *weight*, *name='parametercondition'*)

Bases: *Condition*

A condition that applies a penalty term on some parameters which are optimized during the training process.

Parameters

- **parameter** (*torchphysics.Parameter*) – The parameter that should be optimized.
- **penalty** (*callable*) – A user-defined function that defines a penalty term on the parameters.
- **weight** (*float*) – The weight multiplied with the loss of the penalty during training.
- **name** (*str*) – The name of this condition which will be monitored in logging.

forward(*device='cpu'*, *iteration=None*)

The forward run performed by this condition.

Returns
torch.Tensor

Return type
the loss which should be minimized or monitored during training

training: bool

class torchphysics.problem.conditions.condition.**PeriodicCondition**(*module*, *periodic_interval*, *residual_fn*, *non_periodic_sampler=<torchphysics.problem.s object>*, *error_fn=SquaredError()*, *reduce_fn=<built-in method mean of type object>*, *name='periodiccondition'*, *track_gradients=True*, *data_functions={}*, *parameter=Parameter: {}*, *weight=1.0*)

Bases: *Condition*

A condition that allows to learn dependencies between points at the ends of a given Interval. Can be used e.g. for a variety of periodic boundary conditions.

Parameters

- **module** (*torchphysics.Model*) – The torch module which should be optimized.
- **periodic_interval** (*torchphysics.domains.Interval*) – The interval on which' boundary the periodic (boundary) condition will be set.
- **non_periodic_sampler** (*torchphysics.samplers.PointSampler*) – A sampler that creates the points for the axis that are not defined via the periodic_interval
- **residual_fn** (*callable*) – A user-defined function that computes the residual (unreduced loss) from inputs and outputs of the model, e.g. by using utils.differentialoperators and/or domain.normal. Instead of the name of the axis of the periodic interval, it takes {name}_left and {name}_right as an input. The same holds for all outputs of the network and the results of the data_functions.
- **error_fn** (*callable*) – Function that will be applied to the output of the residual_fn to compute the unreduced loss. Should reduce only along the 2nd (i.e. space-)axis.
- **reduce_fn** (*callable*) – Function that will be applied to reduce the loss to a scalar. Defaults to torch.mean
- **data_functions** (*dict*) – A dictionary of user-defined functions and their names (as keys). Can be used e.g. for right sides in PDEs or functions in boundary conditions.
- **track_gradients** (*bool*) – Whether gradients w.r.t. the inputs should be tracked during training or not. Defaults to true, since this is needed to compute differential operators in PINNs.
- **parameter** (Parameter) – A Parameter that can be used in the residual_fn and should be learned in parallel, e.g. based on data (in an additional DataCondition).
- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.

forward(*device='cpu'*, *iteration=None*)

The forward run performed by this condition.

Returns
torch.Tensor

Return type
the loss which should be minimized or monitored during training

training: bool

class torchphysics.problem.conditions.condition.**SingleModuleCondition**(*module*, *sampler*, *residual_fn*, *error_fn*, *reduce_fn=<built-in method mean of type object>*, *name='singlemodulecondition'*, *track_gradients=True*, *data_functions={}*, *parameter=Parameter: {}*, *weight=1.0*)

Bases: *Condition*

A condition that minimizes the reduced loss of a single module.

Parameters

- **module** (*torchphysics.Model*) – The torch module which should be optimized.
- **sampler** (*torchphysics.samplers.PointSampler*) – A sampler that creates the points in the domain of the residual function, could be an inner or a boundary domain.
- **residual_fn** (*callable*) – A user-defined function that computes the residual (unreduced loss) from inputs and outputs of the model, e.g. by using utils.differentialoperators and/or domain.normal
- **error_fn** (*callable*) – Function that will be applied to the output of the residual_fn to compute the unreduced loss. Should reduce only along the 2nd (i.e. space-)axis.
- **reduce_fn** (*callable*) – Function that will be applied to reduce the loss to a scalar. Defaults to torch.mean
- **data_functions** (*dict*) – A dictionary of user-defined functions and their names (as keys). Can be used e.g. for right sides in PDEs or functions in boundary conditions.
- **track_gradients** (*bool*) – Whether gradients w.r.t. the inputs should be tracked during training or not. Defaults to true, since this is needed to compute differential operators in PINNs.
- **parameter** (Parameter) – A Parameter that can be used in the residual_fn and should be learned in parallel, e.g. based on data (in an additional DataCondition).
- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.

forward(*device='cpu'*, *iteration=None*)

The forward run performed by this condition.

Returns
torch.Tensor

Return type
the loss which should be minimized or monitored during training

training: bool

class torchphysics.problem.conditions.condition.**SquaredError**

Bases: Module

Implements the sum of squared errors in space dimension.

forward(*x*)

Computes the squared error of the input.

Parameters
x (*torch.tensor*) – The values for which the squared error should be computed.

training: bool

2.1.3 torchphysics.problem.conditions.deeponet_condition module

class torchphysics.problem.conditions.deeponet_condition.**DeepONetDataCondition**(*module*, *dataloader*, *norm*, *use_full_dataset=False*, *name='datacondition'*, *weight=1.0*)

Bases: *DataCondition*

A condition that fits a single given module to data (handed through a PyTorch dataloader).

Parameters

- **module** (*torchphysics.Model*) – The torch module which should be fitted to data.
- **dataloader** (*torch.utils.DataLoader*) – A PyTorch dataloader which supplies the iterator to load data-target pairs from some given dataset. Data and target should be handed as points in input or output spaces, i.e. with the correct point object.
- **norm** (*int or 'inf'*) – The 'norm' which should be computed for evaluation. If 'inf', maximum norm will be used. Else, the result will be taken to the n-th potency (without computing the root!)
- **use_full_dataset** (*bool*) – Whether to perform single iterations or compute the error on the whole dataset during forward call. The latter can especially be useful during validation.
- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.

training: bool

class torchphysics.problem.conditions.deeponet_condition.**DeepONetSingleModuleCondition**(*deeponet_model*, *function_set*, *input_sampler*, *residual_fn*, *error_fn*, *reduce_fn=<built-in method mean of type object>*, *name='singlemodule*
track_gradients=True
data_functions={}, *parameter=Parameter: {}*, *weight=1.0*)

Bases: *Condition*

forward(*device='cpu'*, *iteration=None*)

The forward run performed by this condition.

Returns
torch.Tensor

Return type
the loss which should be minimized or monitored during training

training: bool

class torchphysics.problem.conditions.deeponet_condition.**PIDeepONetCondition**(*deeponet_model*, *function_set*, *input_sampler*, *residual_fn*, *name='pinncondition'*, *track_gradients=True*, *data_functions={}*, *parameter=Parameter: {}*, *weight=1.0*)

Bases: *DeepONetSingleModuleCondition*

A condition that minimizes the mean squared error of the given residual, as required in the framework of physics-informed DeepONets [1].

Parameters

- **deeponet_model** (*torchphysics.models.DeepONet*) – The DeepONet-model, consisting of trunk and branch net that should be optimized.
- **function_set** (*torchphysics.domains.FunctionSet*) – A FunctionSet that provides the different input functions for the branch net.
- **input_sampler** (*torchphysics.samplers.PointSampler*) – A sampler that creates the points inside the domain of the residual function, could be an inner or a boundary domain.
- **residual_fn** (*callable*) – A user-defined function that computes the residual (unreduced loss) from inputs and outputs of the model, e.g. by using utils.differentialoperators and/or domain.normal
- **data_functions** (*dict*) – A dictionary of user-defined functions and their names (as keys). Can be used e.g. for right sides in PDEs or functions in boundary conditions.
- **track_gradients** (*bool*) – Whether gradients w.r.t. the inputs should be tracked during training or not. Defaults to true, since this is needed to compute differential operators in PINNs.
- **parameter** (Parameter) – A Parameter that can be used in the residual_fn and should be learned in parallel, e.g. based on data (in an additional DataCondition).
- **name** (*str*) – The name of this condition which will be monitored in logging.
- **weight** (*float*) – The weight multiplied with the loss of this condition during training.

Notes

training: bool

2.2 torchphysics.problem.domains package

Domains handle the geometries of the underlying problems. Every input variable, that appears in the differentialequation has to get a domain, to which it belongs. Different 0D, 1D, 2D and 3D domains are pre implemented. For more complex domains four operations are implemented:

- `Union` $A \cup B$, implemented with: `+`
- `Intersection` $A \cap B$, implemented with: `&`
- `Cut` $A \setminus B$, implemented with: `-`
- `Cartesian product` $A \times B$, implemented with: `*`

It is possible to pass in functions as parameters of most domains. This leads to geometries that can change depending on other variables, e.g. a moving domain in time.

Boolean operations together with cartesian products and parameter-dependencies form an easy-to-use toolbox enable the user to define a large set of relevant domains. In addition, domains can be imported from shapely or *.stl*-files (through *trimesh*).

If you want to solve an inverse problem, the learnable parameters **do not** get a domain! They have to be defined with the torchphysics.model.Parameters class.

2.2.1 Subpackages

torchphysics.problem.domains.domain0D package

Submodules

torchphysics.problem.domains.domain0D.point module

class torchphysics.problem.domains.domain0D.point.**Point**(*space*, *point*)

Bases: *Domain*

Creates a single point at the given coordinates.

Parameters

- **space** (Space) – The space in which this object lays.
- **coord** (*Number, List or callable*) – The coordinate of the point.

__call__(***data*)

Evaluates the domain at the given data.

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns

A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

torchphysics.problem.domains.domain1D package

Submodules

torchphysics.problem.domains.domain1D.interval module

class torchphysics.problem.domains.domain1D.interval.**Interval**(*space*, *lower_bound*, *upper_bound*)

Bases: *Domain*

Creates a Interval of the form [a, b].

Parameters

- **space** (Space) – The space in which this object lays.
- **lower_bound** (*Number or callable*) – The left/lower bound of the interval.
- **upper_bound** (*Number or callable*) – The right/upper bound of the interval.

__call__(***data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns
boundary – The boundary-object of the domain.

Return type
torchphysics.domains.Boundarydomain

property boundary_left

Returns only the left boundary value, useful for the definintion of inital conditions.

property boundary_right

Returns only the left boundary value, useful for the definintion of end conditions.

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns
A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domain1D.interval.**IntervalBoundary**(*domain*)

Bases: *BoundaryDomain*

normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.
- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns
The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type
torch.tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.

- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domain1D.interval.**IntervalSingleBoundaryPoint**(*domain*, *side*, *normal_vec=-1*)

Bases: *BoundaryDomain*

__call__(**data*)
Evaluates the domain at the given data.

normal(*points*, *params=Points: {}*, *device='cpu'*)
Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.
- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns
The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type
torch.tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)
Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points object containing the sampled points.

Return type

Points

torchphysics.problem.domains.domain2D package

Submodules

torchphysics.problem.domains.domain2D.circle module

class torchphysics.problem.domains.domain2D.circle.**Circle**(*space*, *center*, *radius*)

Bases: *Domain*

Class for circles.

Parameters

- **space** (Space) – The space in which this object lays.
- **center** (*array_like or callable*) – The center of the circle, e.g. center = [5,0].
- **radius** (*number or callable*) – The radius of the circle.

__call__(***data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns

boundary – The boundary-object of the domain.

Return type

torchphysics.domains.Boundarydomain

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns

A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domain2D.circle.**CircleBoundary**(*domain*)

Bases: *BoundaryDomain*

normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.
- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns
The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type
torch.tensor

`sample_grid`(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

`sample_random_uniform`(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

torchphysics.problem.domains.domain2D.parallelogram module

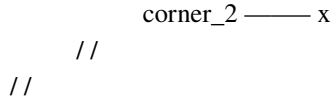
class torchphysics.problem.domains.domain2D.parallelogram.**Parallelogram**(*space*, *origin*, *corner_1*, *corner_2*)

Bases: *Domain*

Class for arbitrary parallelograms, even if time dependet will always stay a parallelogram.

Parameters

- **space** (Space) – The space in which this object lays.
- **origin** (*array_like or callable*) – Three corners of the parallelogram, in the following order



origin —— corner_1

E.g. for the unit square: origin = [0,0], corner_1 = [1,0], corner_2 = [0,1].

- **corner_1** (*array_like or callable*) – Three corners of the parallelogram, in the following order

```
                corner_2 ——— x
          / /
    / /
origin —— corner_1
```

E.g. for the unit square: origin = [0,0], corner_1 = [1,0], corner_2 = [0,1].

- **corner_2** (*array_like or callable*) – Three corners of the parallelogram, in the following order

```
                corner_2 ——— x
          / /
    / /
origin —— corner_1
```

E.g. for the unit square: origin = [0,0], corner_1 = [1,0], corner_2 = [0,1].

__call__(***data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns
boundary – The boundary-object of the domain.

Return type
torchphysics.domains.Boundarydomain

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns
A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domain2D.parallelogram.**ParallelogramBoundary**(*domain*)

Bases: *BoundaryDomain*

normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.
- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns
The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type
torch.tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points object containing the sampled points.

Return type

Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points object containing the sampled points.

Return type

Points

torchphysics.problem.domains.domain2D.shapely_polygon module

class torchphysics.problem.domains.domain2D.shapely_polygon.**ShapelyBoundary**(*domain*)

Bases: *BoundaryDomain*

__call__(**data*)

Evaluates the domain at the given data.

normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.

- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns
The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type
torch.tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domain2D.shapely_polygon.**ShapelyPolygon**(*space*, *vertices=None*, *shapely_polygon=None*)

Bases: *Domain*

Class for polygons. Uses the shapely-package.

Parameters

- **space** (Space) – The space in which this object lays.
- **vertices** (*list of lists, optional*) – The corners/vertices of the polygon. Can be eihter in clockwise or counter- clockwise order.

- **shapely_polygon** (*shapely.geometry.Polygon, optional*) – Instead of defining the corner points, it is also possible to give a already existing shapely.Polygon object.

Note: This class can not be dependent on other variables.

__call__(***data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns
boundary – The boundary-object of the domain.

Return type
torchphysics.domains.Boundarydomain

bounding_box(*device='cpu'*)

Computes the bounds of the domain.

Returns
A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

outline(*device='cpu'*)

Creates a outline of the domain.

Returns
The vertices of the domain. Inner vertices are appended in there own list.

Return type
list of list

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points object containing the sampled points.

Return type

Points

torchphysics.problem.domains.domain2D.triangle module

class torchphysics.problem.domains.domain2D.triangle.**Triangle**(*space*, *origin*, *corner_1*, *corner_2*)

Bases: *Domain*

Class for triangles.

Parameters

- **space** (Space) – The space in which this object lays.
- **origin** (*array_like or callable*) – The three corners of the triangle. The corners have to be ordered counter clockwise, to assure that the normal vectors will point outwards.
- **corner_1** (*array_like or callable*) – The three corners of the triangle. The corners have to be ordered counter clockwise, to assure that the normal vectors will point outwards.
- **corner_2** (*array_like or callable*) – The three corners of the triangle. The corners have to be ordered counter clockwise, to assure that the normal vectors will point outwards.

__call__(***data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns

boundary – The boundary-object of the domain.

Return type

torchphysics.domains.Boundarydomain

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns

A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max,

axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domain2D.triangle.**TriangleBoundary**(*domain*)

Bases: *BoundaryDomain*

normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.

- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns
The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type
torch.tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

torchphysics.problem.domains.domain3D package

Submodules

torchphysics.problem.domains.domain3D.sphere module

class torchphysics.problem.domains.domain3D.sphere.**Sphere**(*space*, *center*, *radius*)

Bases: *Domain*

Class for a sphere.

Parameters

- **space** (Space) – The space in which this object lays.
- **center** (*array_like or callable*) – The center of the sphere, e.g. center = [5, 0, 0].
- **radius** (*number or callable*) – The radius of the sphere.

__call__(**data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns
boundary – The boundary-object of the domain.

Return type
torchphysics.domains.Boundarydomain

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns
A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.

- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domain3D.sphere.**SphereBoundary**(*domain*)

Bases: *BoundaryDomain*

normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.
- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns
The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type
torch.tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

torchphysics.problem.domains.domain3D.trimesh_polyhedron module

class torchphysics.problem.domains.domain3D.trimesh_polyhedron.**TrimeshBoundary**(*domain*)

Bases: *BoundaryDomain*

normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.
- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns
The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type
torch.tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domain3D.trimesh_polyhedron.**TrimeshPolyhedron**(*space*, *vertices=None*, *faces=None*, *file_name=None*, *file_type='stl'*, *tol=1e-06*)

Bases: *Domain*

Class for polygons in 3D. Uses the trimesh-package.

Parameters

- **space** (Space) – The space in which this object lays.
- **vertices** (*list of lists, optional*) – The vertices of the polygon.
- **faces** (*list of lists, optional*) – A list that contains which vetrices have to be connected to create the faces of the polygon. If for example the vertices 1, 2 and 3 have should be connected do: faces = [[1, 2, 3]]
- **file_name** (*str or file-like object, optional*) – A data source to load a existing polygon/mesh.
- **file_type** (*str, optional*) – The file type, e.g. 'stl'. See trimesh.available_formats() for all supported file types.
- **tol** (*number, optional*) – The error tolerance for checking if points at the boundary. And used for projections and slicing the mesh.

Note: This class can not be dependent on other variables.

__call__(***data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns
boundary – The boundary-object of the domain.

Return type
torchphysics.domains.Boundarydomain

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns
A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

export_file(*name_of_file*)

Exports the mesh to a file.

Parameters
name_of_file (*str*) – The name of the file.

project_on_plane(*new_space*, *plane_origin=[0, 0, 0]*, *plane_normal=[0, 0, 1]*)

Projects the polygon on a plane.

Parameters

- **new_space** (Space) – The space in which the projected object should lay.
- **plane_origin** (*array_like, optional*) – The origin of the projection plane.
- **plane_normal** (*array_like, optional*) – The normal vector of the projection plane. It is enough if it points in the direction of normal vector, it does not norm = 1.

Returns
The polygon that is the outline of the projected original mesh on the plane.

Return type
ShapelyPolygon

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

slice_with_plane(*new_space*, *plane_origin=[0, 0, 0]*, *plane_normal=[0, 0, 1]*)

Slices the polygon with a plane.

Parameters

- **new_space** (Space) – The space in which the projected object should lay.
- **plane_origin** (*array_like, optional*) – The origin of the plane.
- **plane_normal** (*array_like, optional*) – The normal vector of the projection plane. It is enough if it points in the direction of normal vector, it does not norm = 1.

Returns
The polygon that is the outline of the projected original mesh on the plane.

Return type
ShapelyPolygon

torchphysics.problem.domains.domainoperations namespace

Submodules

torchphysics.problem.domains.domainoperations.cut module

class torchphysics.problem.domains.domainoperations.cut.**CutBoundaryDomain**(*domain:* CutDomain)

Bases: *BoundaryDomain*

normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.
- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns
The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type
torch.tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domainoperations.cut.**CutDomain**(*domain_a: Domain*, *domain_b: Domain*, *contained=False*)

Bases: *Domain*

Implements the logical cut of two domains.

Parameters

- **domain_a** (Domain) – The first domain.
- **domain_b** (Domain) – The second domain.

__call__(***data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns
boundary – The boundary-object of the domain.

Return type
torchphysics.domains.Boundarydomain

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns
A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.

- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

torchphysics.problem.domains.domainoperations.intersection module

class torchphysics.problem.domains.domainoperations.intersection.**IntersectionBoundaryDomain**(*domain: IntersectionDomain*)

Bases: *BoundaryDomain*

normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.
- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns
The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type
torch.tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domainoperations.intersection.**IntersectionDomain**(*domain_a: Domain*, *domain_b: Domain*)

Bases: *Domain*

Implements the logical intersection of two domains.

Parameters

- **domain_a** (Domain) – The first domain.
- **domain_b** (Domain) – The second domain.

__call__(***data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns
boundary – The boundary-object of the domain.

Return type
torchphysics.domains.Boundarydomain

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns
A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

torchphysics.problem.domains.domainoperations.product module

class torchphysics.problem.domains.domainoperations.product.**ProductDomain**(*domain_a*, *domain_b*)

Bases: *Domain*

The 'cartesian' product of two domains. Additionally supports dependence of domain_a on domain_b, i.e. if the definition of domain_a contains functions of variables in domain_b.space, they are evaluated properly.

Parameters

- **domain_a** (Domain) – The (optionally dependent) first domain.
- **domain_b** (Domain) – The second domain.

__call__(***data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns
boundary – The boundary-object of the domain.

Return type
torchphysics.domains.Boundarydomain

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns
A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

set_bounding_box(*bounds*)

To set the bounds of the domain.

Parameters

bounds (*list*) – The bounding box of the domain. Whereby the lenght of the list has to be two times the domain dimension. And the bounds need to be in the following order: [min_axis_1, max_axis_1, min_axis_2, max_axis_2, ...]

torchphysics.problem.domains.domainoperations.sampler_helper module

This file contains some sample functions for the domain operations. Since Union/Cut/Intersection follow the same idea for sampling for a given number of points.

torchphysics.problem.domains.domainoperations.union module

class torchphysics.problem.domains.domainoperations.union.**UnionBoundaryDomain**(*domain: UnionDomain*)

Bases: *BoundaryDomain*

normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.
- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns

The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type

torch.tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

class torchphysics.problem.domains.domainoperations.union.**UnionDomain**(*domain_a: Domain*, *domain_b: Domain*, *disjoint=False*)

Bases: *Domain*

Implements the logical union of two domains.

Parameters

- **domain_a** (Domain) – The first domain.
- **domain_b** (Domain) – The second domain.

__call__(***data*)

Evaluates the domain at the given data.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns
boundary – The boundary-object of the domain.

Return type
torchphysics.domains.Boundarydomain

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns
A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

sample_grid(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

sample_random_uniform(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

torchphysics.problem.domains.functionsets package

Function sets can be used to sample functions, e.g. in DeepONet.

Submodules

torchphysics.problem.domains.functionsets.functionset module

class torchphysics.problem.domains.functionsets.functionset.**CustomFunctionSet**(*function_space*, *parameter_sampler*, *custom_fn*)

Bases: *FunctionSet*

FunctionSet for an arbitrary function.

Parameters

- **function_space** (*torchphysics.spaces.FunctionSpace*) – The space of which this set of functions belongs to. The inputs and outputs of this FunctionSet are defined by the corresponding values inside the function space.
- **parameter_sampler** (*torchphysics.samplers.PointSampler*) – A sampler that provides additional parameters that can be used to create different kinds of functions. E.g. our FunctionSet consists of Functions like k*x, x is the input variable and k is given through the sampler.

  During each training iteration will call the parameter_sampler to sample new parameters. For each parameter a function will be created and the input batch of functions will be of the same length as the sampled parameters.
- **custom_fn** (*callable*) – A function that describes the FunctionSet. The input of the functions can include the variables of the function_space.input_space and the parameters from the parameter_sampler.

class torchphysics.problem.domains.functionsets.functionset.**FunctionSet**(*function_space*, *parameter_sampler*)

Bases: object

A set of functions that can supply samples from a function space.

Parameters

- **function_space** (*torchphysics.spaces.FunctionSpace*) – The space of which this set of functions belongs to. The inputs and outputs of this FunctionSet are defined by the corresponding values inside the function space.
- **parameter_sampler** (*torchphysics.samplers.PointSampler*) – A sampler that provides additional parameters that can be used to create different kinds of functions. E.g. our FunctionSet consists of Functions like k*x, x is the input variable and k is given through the sampler.

  During each training iteration will call the parameter_sampler to sample new parameters. For each parameter a function will be created and the input batch of functions will be of the same length as the sampled parameters.

__add__(*other*)

Combines two function sets.

Notes

When parameters are sampled, will sample them from both sets. Creates a batch of functions consisting of the batch of each set. (Length of the batches will be added)

__len__()

Returns the amount of functions sampled in a single call to sample_params.

create_function_batch(*points*)

Evaluates the underlying function object to create a batch of discrete function samples.

Parameters

points (*torchphysics.spaces.Points*) – The input points, where we want to evaluate a set of functions.

Returns

The batch of discrete function samples. The underlying tensor is of the shape: [len(self), len(points), self.function_space.output_space.dim]

Return type
torchphysics.spaces.Points

`sample_params`(*device='cpu'*)

Samples parameters of the function space.

Parameters
device (*str, optional*) – The device, where the parameters should be created. Default is 'cpu'.

Notes

We save the sampled parameters internally, so that we can use them multiple times. Since given a parameter we still have a continuous representation of the underlying function types. When the functions should be evaluated at some input points, we just have to create the meshgrid of parameters and points.

`class torchphysics.problem.domains.functionsets.functionset.FunctionSetCollection`(*function_sets*)

Bases: *FunctionSet*

Collection of multiple FunctionSets. Used for the additions of different FunctionSets.

Parameters
function_sets (*list, tuple*) – A list/tuple of FunctionSets.

`__add__`(*other*)

Combines two function sets.

Notes

When parameters are sampled, will sample them from both sets. Creates a batch of functions consisting of the batch of each set. (Length of the batches will be added)

`__len__`()

Returns the amount of functions sampled in a single call to sample_params.

`create_function_batch`(*points*)

Evaluates the underlying function object to create a batch of discrete function samples.

Parameters
points (*torchphysics.spaces.Points*) – The input points, where we want to evaluate a set of functions.

Returns
The batch of discrete function samples. The underlying tensor is of the shape: [len(self), len(points), self.function_space.output_space.dim]

Return type
torchphysics.spaces.Points

`sample_params`(*device='cpu'*)

Samples parameters of the function space.

Parameters
device (*str, optional*) – The device, where the parameters should be created. Default is 'cpu'.

Notes

We save the sampled parameters internally, so that we can use them multiple times. Since given a parameter we still have a continuous representation of the underlying function types. When the functions should be evaluated at some input points, we just have to create the meshgrid of parameters and points.

2.2.2 Submodules

2.2.3 torchphysics.problem.domains.domain module

class torchphysics.problem.domains.domain.**BoundaryDomain**(*domain*)

Bases: *Domain*

The parent class for all built-in boundaries. Can be used just like the main Domain class.

Parameters

domain (Domain) – The domain of which this object is the boundary.

__call__(***data*)

Evaluates the domain at the given data.

bounding_box(*params=Points: {}*, *device='cpu'*)

Computes the bounds of the domain.

Returns

A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type

tensor

abstract normal(*points*, *params=Points: {}*, *device='cpu'*)

Computes the normal vector at each point in points.

Parameters

- **points** (*torch.tensor or torchphysics.problem.Points*) – Different points for which the normal vector should be computed. The points should lay on the boundary of the domain, to get correct results. E.g in 2D: points = Points(torch.tensor([[2, 4], [9, 6], ...]), R2(...))
- **params** (*dict or torchphysics.problem.Points, optional*) – Additional parameters that are maybe needed to evaluate the domain.
- **device** (*str, optional*) – The device on which the points should be created. Default is 'cpu'.

Returns

The tensor is of the shape (len(points), self.dim) and contains the normal vector at each entry from points.

Return type

torch.tensor

class torchphysics.problem.domains.domain.**Domain**(*space*, *dim=None*)

Bases: object

The parent class for all built-in domains.

Parameters

- **space** (*torchphysiscs.spaces.Space*) – The space in which this object lays.
- **dim** (*int, optional*) – The dimension of this domain. (if not specified, implicit given through the space)

__add__(*other*)

Creates the union of the two input domains.

Parameters
other (Domain) – The other domain that should be united with the domain. Has to be of the same dimension.

__and__(*other*)

Creates the intersection of the two input domains.

Parameters
other (Domain) – The other domain that should be intersected with the domain. Has to lie in the same space.

__call__(***data*)

Evaluates the domain at the given data.

__contains__(*points*)

Checks for every point in points if it lays inside the domain.

Parameters
points (*torchphysics.problem.Points*) – A Points object that should be checked.

Returns
A boolean Tensor of the shape (len(points), 1) where every entry contains true if the point was inside or false if not.

Return type
torch.Tensor

__mul__(*other*)

Creates the cartesian product of this domain and another domain.

Parameters
other (Domain) – The other domain to create the cartesian product with. Should lie in a disjoint space.

__sub__(*other*)

Creates the cut of domain other from self.

Parameters
other (Domain) – The other domain that should be cut off the domain. Has to be of the same dimension.

property boundary

Returns the boundary of this domain. Does not work on boundaries itself, e.g. Circle.boundary.boundary throws an error.

Returns
boundary – The boundary-object of the domain.

Return type
torchphysics.domains.Boundarydomain

abstract bounding_box(*params=Points: {}, device='cpu'*)

Computes the bounds of the domain.

Returns
A torch.Tensor with the length of 2*self.dim. It has the form [axis_1_min, axis_1_max, axis_2_min, axis_2_max, ...], where min and max are the minimum and maximum value that the domain reaches in each dimension-axis.

Return type
tensor

`compute_n_from_density`(*d*, *params*)

Transforms a given point density to a number of points, since all methods from PyTorch only work with a given number.

`len_of_params`(*params*)

Finds the number of params, for which points should be sampled.

`abstract sample_grid`(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates an equdistant grid in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

`abstract sample_random_uniform`(*n=None*, *d=None*, *params=Points: {}*, *device='cpu'*)

Creates random uniformly distributed points in the domain.

Parameters

- **n** (*int, optional*) – The number of points that should be created.
- **d** (*float, optional*) – The density of points that should be created, if n is not defined.
- **params** (*torchphysics.problem.Points, optional*) – Additional paramters that are maybe needed to evaluate the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points object containing the sampled points.

Return type
Points

`set_necessary_variables`(**domain_params*)

Registers the variables/spaces that this domain needs to be properly defined

`set_volume`(*volume*)

Set the volume of the given domain.

Parameters
volume (*number or callable*) – The volume of the domain. Can be a function if the volume changes depending on other variables.

Notes

For all basic domains the volume (and surface) are implemented. But if the given domain has a complex shape or is dependent on other variables, the volume can only be approixmated. Therefore one can set here a exact expression for the volume, if known.

transform_to_user_functions(**domain_params*)

Transforms all parameters that define a given domain to a UserFunction. This enables that the domain can dependt on other variables.

Parameters
***domain_params** (*callables, lists, arrays, tensors or numbers*) – The parameters that define a domain.

volume(*params=Points: {}*, *device='cpu'*)

Computes the volume of the current domain.

Parameters
params (*torchphysics.problem.Points, optional*) – Additional paramters that are needed to evaluate the domain.

Returns
volume – Returns the volume of the domain. If dependent on other parameters, the value will be returned as tensor with the shape (len(params), 1). Where each row corresponds to the volume of the given values in the params row.

Return type
torch.tensor

2.3 torchphysics.models package

Contains different PyTorch models which can be trained to approximate the solution of a differential equation.

Additional basic network structures are implemented, meant to stabilize and speed up the trainings process. (adaptive weights, normalization layers)

If different models for different parts of the differential equation should be applied, this can be achieved by using the classes torchphysics.models.Sequential and torchphysics.models.Parallel.

Here you also find the parameters that can be learned in inverse problems.

2.3.1 Subpackages

torchphysics.models.deeponet namespace

Submodules

torchphysics.models.deeponet.deeponet module

class torchphysics.models.deeponet.deeponet.**DeepONet**(*trunk_net*, *branch_net*)

Bases: *Model*

Implementation of the architecture used in the DeepONet paper [1]. Consists of two single neural networks. One for the inputs of the function space (branch net) and one for the inputs of the variables (trunk net).

Parameters

- **trunk_net** (*torchphysics.models.TrunkNet*) – The neural network that will get the space/time/... variables as an input.
- **branch_net** (*torchphysics.models.BranchNet*) – The neural network that will get the function variables as an input.

Notes

The number of output neurons in the branch and trunk net have to be the same!

fix_branch_input(*function*, *device='cpu'*)

Fixes the branch net for a given function. this function will then be used in every following forward call. To set a new function just call this method again.

Parameters

- **function** (*callable, torchphysics.domains.FunctionSet*) – The function(s) for which the branch should be evaluaded.
- **device** (*str, optional*) – The device where the data lays. Default is 'cpu'.

forward(*trunk_inputs*, *branch_inputs=None*, *device='cpu'*)

Apply the network to the given inputs.

Parameters

- **trunk_inputs** (*torchphysics.spaces.Points*) – The inputs for the trunk net.
- **branch_inputs** (*callable, torchphysics.domains.FunctionSet, optional*) – The function(s) for which the branch should be evaluaded. If no input is given, the branch net has to be fixed before hand!
- **device** (*str, optional*) – The device where the data lays. Default is 'cpu'.

Returns
A point object containing the output.

Return type
torchphysics.spaces.Points

training: bool

torchphysics.models.deeponet.subnets module

class torchphysics.models.deeponet.subnets.**BranchNet**(*function_space*, *output_space*, *output_neurons*, *discretization_sampler*)

Bases: *Model*

A neural network that can be used inside a DeepONet-model.

Parameters

- **function_space** (Space) – The space of functions that can be put in this network.

- **output_space** (Space) – The space of the points that should be returned by the parent DeepONet-model.
- **output_neurons** (*int*) – The number of output neurons. These neurons will only be used internally. Will be multiplied my the dimension of the output space, so each dimension will have the same number of intermediate neurons. The final output of the DeepONet-model will be in the dimension of the output space.
- **discretization_sampler** (*torchphysics.sampler*) – A sampler that will create the points at which the input functions should evaluated, to create a discrete input for the network. The number of input neurons will be equal to the number of sampled points. Therefore, the sampler should always return the same number of points!

fix_input(*function*, *device='cpu'*)

Fixes the branch net for a given function. The branch net will be evaluated for the given function and the output saved in `current_out`.

Parameters

- **function** (*callable, torchphysics.domains.FunctionSet*) – The function(s) for which the network should be evaluaded.
- **device** (*str, optional*) – The device where the data lays. Default is 'cpu'.

Notes

To overwrite the data `current_out` (the fixed function) just call `.fix_input` again with a new function.

abstract forward(*discrete_function_batch*, *device='cpu'*)

Evaluated the network at a given function batch. Should not be called directly, rather use the method `.fix_input`.

Parameters

- **discrete_function_batch** (*tensor*) – A tensor of discrete function values to evaluate the model.
- **device** (*str, optional*) – The device where the data lays. Default is 'cpu'.

Notes

Will, in general, not return anything. The output of the network will be saved internally to be used multiple times.

training: bool

class torchphysics.models.deeponet.subnets.**FCBranchNet**(*function_space*, *output_space*, *output_neurons*, *discretization_sampler*, *hidden=(20, 20, 20)*, *activations=Tanh()*, *xavier_gains=1.6666666666666667*)

Bases: *BranchNet*

A neural network that can be used inside a DeepONet-model.

Parameters

- **function_space** (Space) – The space of functions that can be put in this network.
- **output_space** (Space) – The space of the points that should be returned by the parent DeepONet-model.

- **output_neurons** (*int*) – The number of output neurons. These neurons will only be used internally. The final output of the DeepONet-model will be in the dimension of the output space.
- **discretization_sampler** (*torchphysics.sampler*) – A sampler that will create the points at which the input functions should evaluated, to create a discrete input for the network. The number of input neurons will be equal to the number of sampled points. Therefore, the sampler should always return the same number of points!
- **hidden** (*list or tuple*) – The number and size of the hidden layers of the neural network. The lenght of the list/tuple will be equal to the number of hidden layers, while the i-th entry will determine the number of neurons of each layer.
- **activations** (*torch.nn or list, optional*) – The activation functions of this network. Deafult is nn.Tanh().
- **xavier_gains** (*float or list, optional*) – For the weight initialization a Xavier/Glorot algorithm will be used. Default is 5/3.

forward(*discrete_function_batch*)

Evaluated the network at a given function batch. Should not be called directly, rather use the method `.fix_input`.

Parameters

- **discrete_function_batch** (*tensor*) – A tensor of discrete function values to evaluate the model.
- **device** (*str, optional*) – The device where the data lays. Default is 'cpu'.

Notes

Will, in general, not return anything. The output of the network will be saved internally to be used multiple times.

training: bool

class torchphysics.models.deeponet.subnets.**FCTrunkNet**(*input_space*, *output_space*, *output_neurons*, *hidden=(20, 20, 20)*, *activations=Tanh()*, *xavier_gains=1.6666666666666667*)

Bases: *TrunkNet*

A fully connected neural networks that can be used inside a DeepONet.

Parameters

- **input_space** (Space) – The space of the points the can be put into this model.
- **output_space** (Space) – The space of the points that should be returned by the parent DeepONet-model.
- **output_neurons** (*int*) – The number of output neurons. These neurons will only be used internally. The final output of the DeepONet-model will be in the dimension of the output space.
- **hidden** (*list or tuple*) – The number and size of the hidden layers of the neural network. The lenght of the list/tuple will be equal to the number of hidden layers, while the i-th entry will determine the number of neurons of each layer.
- **activations** (*torch.nn or list, optional*) – The activation functions of this network. Deafult is nn.Tanh().

- **xavier_gains** (*float or list, optional*) – For the weight initialization a Xavier/Glorot algorithm will be used. Default is 5/3.

forward(*points*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class torchphysics.models.deeponet.subnets.**TrunkNet**(*input_space*, *output_space*, *output_neurons*)

Bases: *Model*

A neural network that can be used inside a DeepONet-model.

Parameters

- **input_space** (Space) – The space of the points that can be put into this model.
- **output_space** (Space) – The number of output neurons. These neurons will only be used internally. The final output of the DeepONet-model will be in the dimension of the output space.
- **output_neurons** (*int*) – The number of output neurons. Will be multiplied my the dimension of the output space, so each dimension will have the same number of intermediate neurons.

training: bool

2.3.2 Submodules

2.3.3 torchphysics.models.activation_fn module

class torchphysics.models.activation_fn.**AdaptiveActivationFunction**(*activation_fn*, *inital_a=1.0*, *scaling=1.0*)

Bases: `Module`

Implementation of the adaptive activation functions used in [1]. Will create activations of the form: activation_fn(scaling*a * x), where activation_fn is an arbitrary function, a is the additional hyperparameter and scaling is an additional scaling factor.

Parameters

- **activation_fn** (*torch.nn.module*) – The underlying function that should be used for the activation.
- **inital_a** (*float, optional*) – The inital value for the adaptive parameter a. Changes the 'slop' of the underlying function. Default is 1.0
- **scaling** (*float, optional*) – An additional scaling factor, such that the 'a' only has to learn only small values. Will stay fixed while training. Default is 1.0

Notes

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: *bool*

class torchphysics.models.activation_fn.**ReLUn**(*n*)

Bases: `Module`

Implementation of a smoother version of ReLU, in the form of relu(x)**n.

Parameters

n (*float*) – The power to which the inputs should be rasied before appplying the rectified linear unit function.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: *bool*

class torchphysics.models.activation_fn.**Sinus**

Bases: `Module`

Implementation of a sinus activation.

forward(*input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: *bool*

class torchphysics.models.activation_fn.**relu_n**(**args*, ***kwargs*)

Bases: `Function`

static backward(*ctx*, *grad_output*)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by as many outputs as the *forward()* returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to *forward()*. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., *backward()* will have `ctx.needs_input_grad[0] = True` if the first input to *forward()* needs gradient computated w.r.t. the output.

static forward(*ctx*, *x*, *n*)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on *ctx* (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with `ctx.save_for_backward()` if they are intended to be used in `backward` (equivalently, `vjp`) or `ctx.save_for_forward()` if they are intended to be used for in `jvp`.

2.3.4 torchphysics.models.deepritz module

class torchphysics.models.deepritz.**DeepRitzNet**(*input_space*, *output_space*, *width*, *depth*)

Bases: *Model*

Implementation of the architecture used in the Deep Ritz paper [1]. Consists of fully connected layers and residual connections.

Parameters

- **input_space** (Space) – The space of the points the can be put into this model.
- **output_space** (Space) – The space of the points returned by this model.
- **width** (*int*) – The width of the used hidden fully connected layers.
- **depth** (*int*) – The amount of subsequent residual blocks.

Notes

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

2.3.5 torchphysics.models.fcn module

class torchphysics.models.fcn.**FCN**(*input_space*, *output_space*, *hidden=(20, 20, 20)*, *activations=Tanh()*, *xavier_gains=1.6666666666666667*)

Bases: *Model*

A simple fully connected neural network.

Parameters

- **input_space** (Space) – The space of the points the can be put into this model.
- **output_space** (Space) – The space of the points returned by this model.
- **hidden** (*list or tuple*) – The number and size of the hidden layers of the neural network. The lenght of the list/tuple will be equal to the number of hidden layers, while the i-th entry will determine the number of neurons of each layer. E.g hidden = (10, 5) -> 2 layers, with 10 and 5 neurons.
- **activations** (*torch.nn or list, optional*) – The activation functions of this network. If a single function is passed as an input, will use this function for each layer. If a list is used, will use the i-th entry for i-th layer. Deafult is nn.Tanh().
- **xavier_gains** (*float or list, optional*) – For the weight initialization a Xavier/Glorot algorithm will be used. The gain can be specified over this value. Default is 5/3.

forward(*points*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

2.3.6 torchphysics.models.model module

class torchphysics.models.model.**AdaptiveWeightLayer**(*n*)

Bases: `Module`

Adds adaptive weights to the non-reduced loss. The weights are maximized by reversing the gradients, similar to the idea in [1]. Should currently only be used with fixed points.

Parameters

n (*int*) – The amount of sampled points in each batch.

Notes

class GradReverse(**args*, ***kwargs*)

Bases: Function

static backward(*ctx*, *grad_output*)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by as many outputs as the *forward()* returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to *forward()*. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute ctx.needs_input_grad as a tuple of booleans representing whether each input needs gradient. E.g., *backward()* will have ctx.needs_input_grad[0] = True if the first input to *forward()* needs gradient computated w.r.t. the output.

static forward(*ctx*, *x*)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on *ctx* (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with ctx.save_for_backward() if they are intended to be used in backward (equivalently, vjp) or ctx.save_for_forward() if they are intended to be used for in jvp.

forward(*points*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

classmethod grad_reverse(*x*)

training: bool

class torchphysics.models.model.Model(*input_space*, *output_space*)

Bases: Module

Neural networks that can be trained to fulfill user-defined conditions.

Parameters

- **input_space** (Space) – The space of the points the can be put into this model.
- **output_space** (Space) – The space of the points returned by this model.

training: bool

class torchphysics.models.model.**NormalizationLayer**(*domain*)

Bases: *Model*

A first layer that scales a domain to the range (-1, 1)^domain.dim, since this can improve convergence during training.

Parameters

domain (Domain) – The domain from which this layer expects sampled points. The layer will use its bounding box to compute the normalization factors.

forward(*points*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class torchphysics.models.model.**Parallel**(**models*)

Bases: *Model*

A model that wraps multiple models which should be applied in parallel.

Parameters

***models** – The models that should be evaluated parallel. The evaluation happens in the order that the models are passed in. The outputs of the models will be concatenated. The models are not allowed to have the same output spaces, but can have the same input spaces.

forward(*points*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class torchphysics.models.model.**Sequential**(**models*)

Bases: *Model*

A model that wraps multiple models which should be applied sequentially.

Parameters

***models** – The models that should be evaluated sequentially. The evaluation happens in the order that the models are passed in. To work correcty the output of the i-th model has to fit the input of the i+1-th model.

forward(*points*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

2.3.7 torchphysics.models.parameter module

class torchphysics.models.parameter.**Parameter**(*init*, *space*, ***kwargs*)

Bases: *Points*

A parameter that is part of the problem and can be learned during training.

Parameters

- **init** (*number, list, array or tensor*) – The inital guess for the parameter.
- **space** (*torchphysics.problem.spaces.Space*) – The Space to which this parameter belongs. Essentially defines the shape of the parameter, e.g for a single number use R1.

Notes

To use these Parameters during training they have to passed on to the used condition. If many different parameters are used they have to be connected over .join(), see the Points-Class for the exact usage.

If the domains itself should depend on some parameters or the solution sholud be learned for different parameter values, this class should NOT be used. These parameters are mostly meant for inverse problems. Instead, the parameters have to be defined with their own domain and samplers.

2.3.8 torchphysics.models.qres module

class torchphysics.models.qres.**QRES**(*input_space*, *output_space*, *hidden=(20, 20, 20)*, *activations=Tanh()*, *xavier_gains=1.6666666666666667*)

Bases: *Model*

Implements the quadratic residual networks from [1]. Instead of a linear layer, a quadratic layer W_1*x (*) W_2*x + W_1*x + b will be used. Here (*) means the hadamard product of two vectors (elementwise multiplication).

Parameters

- **input_space** (Space) – The space of the points the can be put into this model.
- **output_space** (Space) – The space of the points returned by this model.
- **hidden** (*list or tuple*) – The number and size of the hidden layers of the neural network. The lenght of the list/tuple will be equal to the number of hidden layers, while the i-th entry will determine the number of neurons of each layer. E.g hidden = (10, 5) -> 2 layers, with 10 and 5 neurons.
- **activations** (*torch.nn or list, optional*) – The activation functions of this network. If a single function is passed as an input, will use this function for each layer. If a list is used, will use the i-th entry for i-th layer. Deafult is nn.Tanh().

- **xavier_gains** (*float or list, optional*) – For the weight initialization a Xavier/Glorot algorithm will be used. The gain can be specified over this value. Default is 5/3.

Notes

forward(*points*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class torchphysics.models.qres.**Quadratic**(*in_features*, *out_features*, *xavier_gains*)

Bases: `Module`

Implements a quadratic layer of the form: W_1*x (*) W_2*x + W_1*x + b. Here (*) means the hadamard product of two vectors (elementwise multiplication). W_1, W_2 are weight matrices and b is a bias vector.

Parameters

- **in_features** (*int*) – size of each input sample.
- **out_features** – size of each output sample.
- **xavier_gains** (*float or list*) – For the weight initialization a Xavier/Glorot algorithm will be used. The gain can be specified over this value. Default is 5/3.

forward(*points*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

property in_features

property out_features

training: bool

2.4 torchphysics.problem.samplers package

Objects that sample points in a given domain. These objects handle the creation of training and validation points in the underlying geometries. In general they get the following inputs:

- **domain**: the domain in which the points should be created. If you want to create points at the boundary of a domain, use domain.boundary as an input argument.
- **n_points** or **density**: to set the number of wanted points. Either a fixed number can be chosen or the density of the points.
- **filter**: A function that filters out special points, for example for local boundary conditions.

The default behavior of each sampler is, that in each iteration of the trainings process new points are created and used. If this is not desired, not useful (grid sampling) or not efficient (e.g. really complex domains) one can make every sampler `static`.

Instead of creating the Cartesian product of different domains it is also possible to create the product of different samplers. For some sampling strategies, this is required, e.g. point grids. If the product of the domains is created, the sampler will create points in the new domain. If the product of samplers is created, first every sampler will create points in its own domain and afterwards the product will create a meshgrid of the points. Therefore, the points of the product of samplers has in general more *correlation*, than the points of the domain product.

2.4.1 Submodules

2.4.2 torchphysics.problem.samplers.data_samplers module

File with samplers that handle external created data. E.g. measurements or validation data computed with other methods.

class torchphysics.problem.samplers.data_samplers.**DataSampler**(*points*)

Bases: *PointSampler*

A sampler that processes external created data points.

Parameters

points (*torchphysics.spaces.points or dict*) – The data points that this data sampler should pass to a condition. Either already a torchphysics.spaces.points object or in form of dictionary like: {'x': tensor_for_x, 't': tensor_for_t, ...}. For the dicitionary all tensor need to have the same batch dimension.

sample_points(*params=Points: {}*, *device='cpu'*)

The method that creates the points. Also implemented in all child classes.

Parameters

- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type

Points

2.4.3 torchphysics.problem.samplers.grid_samplers module

File with samplers that create points with some kind of ordered structure.

class torchphysics.problem.samplers.grid_samplers.**ExponentialIntervalSampler**(*domain*, *n_points*, *exponent*)

Bases: *PointSampler*

Will sample non equdistant grid points in the given interval. This works only on intervals!

Parameters

- **domain** (*torchphysics.domain.Interval*) – The Interval in which the points should be sampled.
- **n_points** (*int*) – The number of points that should be sampled.
- **exponent** (*Number*) – Determines how non equdistant the points are and at which corner they are accumulated. They are computed with a grid in [0, 1] and then transformed with the exponent and later scaled/translated:

  exponent < 1: More points at the upper bound.
  points = 1 - x**(1/exponent)

  exponent > 1: More points at the lower bound.
  points = x**(exponent)

sample_points(*params=Points: {}*, *device='cpu'*)

The method that creates the points. Also implemented in all child classes.

Parameters

- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type

Points

class torchphysics.problem.samplers.grid_samplers.**GridSampler**(*domain*, *n_points=None*, *density=None*, *filter_fn=None*)

Bases: *PointSampler*

Will sample a equidistant point grid in the given domain.

Parameters

- **domain** (*torchphysics.domain.Domain*) – The domain in which the points should be sampled.
- **n_points** (*int, optional*) – The number of points that should be sampled.
- **density** (*float, optional*) – The desiered density of the created points.
- **filter_fn** (*callable, optional*) – A function that restricts the possible positions of sample points. A point that is allowed should return True, therefore a point that should be removed must return false. The filter has to be able to work with a batch of inputs. The Sampler will use a rejection sampling to find the right amount of points.

2.4.4 torchphysics.problem.samplers.plot_samplers module

Samplers for plotting and animations of model outputs.

class torchphysics.problem.samplers.plot_samplers.**AnimationSampler**(*plot_domain*, *animation_domain*, *frame_number*, *n_points=None*, *density=None*, *device='cpu'*, *data_for_other_variables={}*)

Bases: *PlotSampler*

A sampler that creates points for an animation.

Parameters

- **plot_domain** (Domain) – The domain over which the model/function should later be plotted. Will create points inside and at the boundary of the domain.
- **animation_domain** (Interval) – The variable over which the animation should be created, e.g a time-interval.
- **frame_number** (*int*) – The number of frames that should be used for the animation. This equals the number of points that will be created in the animation_domain.
- **n_points** (*int, optional*) – The number of points that should be used for the plot domain.
- **density** (*float, optional*) – The desiered density of the created points, in the plot domain.
- **device** (*str or torch device, optional*) – The device of the model/function.
- **data_for_other_variables** (*dict, optional*) – Since the animation will only evaluate the model at specific points, the values for all other variables are needed. E.g. {'D' : [1,2], ...}

property **animation_key**

Retunrs the name of the animation variable

property **plot_domain_constant**

Returns if the plot domain is a constant domain or changes with respect to other variables.

sample_animation_points()

Samples points out of the animation domain, e.g. time interval.

sample_plot_domain_points(*animation_points*)

Samples points in the plot domain, e.g. space.

class torchphysics.problem.samplers.plot_samplers.**PlotSampler**(*plot_domain*, *n_points=None*, *density=None*, *device='cpu'*, *data_for_other_variables={}*)

Bases: *PointSampler*

A sampler that creates a point grid over a domain (including the boundary). Only used for plotting,

Parameters

- **plot_domain** (Domain) – The domain over which the model/function should later be plotted. Will create points inside and at the boundary of the domain.
- **n_points** (*int, optional*) – The number of points that should be used for the plot.
- **density** (*float, optional*) – The desiered density of the created points.

- **device** (*str or torch device, optional*) – The device of the model/function.
- **data_for_other_variables** (*dict or torchphysics.spaces.Points, optional*) – Since the plot will only evaluate the model at a specific point, the values for all other variables are needed. E.g. {'t' : 1, 'D' : [1,2], ...}

Notes

Can also be used to create your own PlotSampler. By either changing the used sampler after the initialization (self.sampler=...) or by creating your own class that inherits from PlotSampler.

construct_sampler()

Construct the sampler which is used in the plot. Can be overwritten to include your own points structure.

sample_points(*params=Points: {}*, *device='cpu'*)

Creates the points for the plot. Does not need additional arguments, since they were set in the init.

set_data_for_other_variables(*data_for_other_variables*)

Sets the data for all other variables. Essentially copies the values into a correct tensor.

transform_data_to_torch(*data_for_other_variables*)

Transforms all inputs to a torch.tensor.

2.4.5 torchphysics.problem.samplers.random_samplers module

File with samplers that create random distributed points.

class torchphysics.problem.samplers.random_samplers.**AdaptiveRandomRejectionSampler**(*domain*, *n_points=None*, *density=None*, *filter_fn=None*)

Bases: *AdaptiveSampler*

An adaptive sampler that creates more points in regions with high loss. During sampling, points with high loss are more likely to be kept for the next iteration, while points with small loss are regarded and resampled (random) uniformly in the whole domain.

Parameters

- **domain** (*torchphysics.domain.Domain*) – The domain in which the points should be sampled.
- **n_points** (*int, optional*) – The number of points that should be sampled.
- **density** (*float, optional*) – The desired initial (and average) density of the created points, actual density will change loccally during iterations.
- **filter** (*callable, optional*) – A function that restricts the possible positions of sample points. A point that is allowed should return True, therefore a point that should be removed must return False. The filter has to be able to work with a batch of inputs. The Sampler will use a rejection sampling to find the right amount of points.

sample_points(*unreduced_loss=None*, *params=Points: {}*, *device='cpu'*)

Extends the sample methode of the parent class. Also requieres the unreduced loss of the previous iteration to create the new points.

Parameters

- **unreduced_loss** (*torch.tensor*) – The tensor containing the loss of each training point in the previous iteration.
- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type
Points

class torchphysics.problem.samplers.random_samplers.**AdaptiveThresholdRejectionSampler**(*domain*, *resample_ratio*, *n_points=None*, *density=None*, *filter_fn=None*)

Bases: *AdaptiveSampler*

An adaptive sampler that creates more points in regions with high loss. During sampling, points with loss larger than ´min(loss)+resample_ratio*(max(loss)-min(loss))´ are kept for the next iteration, while points with small loss are regarded and resampled (random) uniformly in the whole domain.

Parameters

- **domain** (*torchphysics.domain.Domain*) – The domain in which the points should be sampled.
- **resample_ratio** (*float*) – During sampling, points with loss larger than ´min(loss)+resample_ratio*(max(loss)-min(loss))´ are kept for the next iteration, while points with small loss are regarded and resampled (random) uniformly in the whole domain.
- **n_points** (*int, optional*) – The number of points that should be sampled.
- **density** (*float, optional*) – The desired initial (and average) density of the created points, actual density will change loccally during iterations.
- **filter** (*callable, optional*) – A function that restricts the possible positions of sample points. A point that is allowed should return True, therefore a point that should be removed must return False. The filter has to be able to work with a batch of inputs. The Sampler will use a rejection sampling to find the right amount of points.

sample_points(*unreduced_loss=None*, *params=Points: {}*, *device='cpu'*)

Extends the sample methode of the parent class. Also requieres the unreduced loss of the previous iteration to create the new points.

Parameters

- **unreduced_loss** (*torch.tensor*) – The tensor containing the loss of each training point in the previous iteration.
- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.

- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type
Points

class torchphysics.problem.samplers.random_samplers.**GaussianSampler**(*domain*, *n_points*, *mean*, *std*)

Bases: *PointSampler*

Will sample normal/gaussian distributed points in the given domain. Only works for the inner part of a domain, not the boundary!

Parameters

- **domain** (*torchphysics.domain.Domain*) – The domain in which the points should be sampled.
- **n_points** (*int*) – The number of points that should be sampled.
- **mean** (*list, array or tensor*) – The center/mean of the distribution. Has to fit the dimension of the given domain.
- **std** (*number*) – The standard deviation of the distribution.

class torchphysics.problem.samplers.random_samplers.**LHSSampler**(*domain*, *n_points*)

Bases: *PointSampler*

Will create a simple latin hypercube sampling [1] in the given domain. Only works for the inner part of a domain, not the boundary!

Parameters

- **domain** (*torchphysics.domain.Domain*) – The domain in which the points should be sampled.
- **n_points** (*int*) – The number of points that should be sampled.

Notes

A bounding box is used tp create the lhs-points in the domain. Points outside will be rejected and additional random uniform points will be added to get a total number of n_points. .. [1] https://en.wikipedia.org/wiki/Latin_hypercube_sampling

class torchphysics.problem.samplers.random_samplers.**RandomUniformSampler**(*domain*, *n_points=None*, *density=None*, *filter_fn=None*)

Bases: *PointSampler*

Will sample random uniform distributed points in the given domain.

Parameters

- **domain** (*torchphysics.domain.Domain*) – The domain in which the points should be sampled.
- **n_points** (*int, optional*) – The number of points that should be sampled.
- **density** (*float, optional*) – The desiered density of the created points.

- **filter** (*callable, optional*) – A function that restricts the possible positions of sample points. A point that is allowed should return True, therefore a point that should be removed must return False. The filter has to be able to work with a batch of inputs. The Sampler will use a rejection sampling to find the right amount of points.

2.4.6 torchphysics.problem.samplers.sampler_base module

The basic structure of every sampler and all sampler 'operations'.

class torchphysics.problem.samplers.sampler_base.**AdaptiveSampler**(*n_points=None*, *density=None*, *filter_fn=None*)

Bases: *PointSampler*

A sampler that requires a current loss for every point of the last sampled set of points.

sample_points(*unreduced_loss*, *params=Points: {}*, *device='cpu'*)

Extends the sample methode of the parent class. Also requieres the unreduced loss of the previous iteration to create the new points.

Parameters

- **unreduced_loss** (*torch.tensor*) – The tensor containing the loss of each training point in the previous iteration.
- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type

Points

class torchphysics.problem.samplers.sampler_base.**AppendSampler**(*sampler_a*, *sampler_b*)

Bases: *PointSampler*

A sampler that appends the output of two samplers behind each other. Essentially calling torch.coloumn_stack for the data points.

Parameters

- **sampler_a** (PointSampler) – The two PointSamplers that should be connected. Both Samplers should create the same number of points.
- **sampler_b** (PointSampler) – The two PointSamplers that should be connected. Both Samplers should create the same number of points.

__len__()

Returns the number of points that the sampler will create or has created.

Note: This can be only called if the number of points is set with `n_points`. Elsewise the the number can only be known after the first call to `sample_points` methode or may even change after each call. If you know the number of points yourself, you can set this with `.set_length`.

sample_points(*params=Points: {}*, *device='cpu'*)

The method that creates the points. Also implemented in all child classes.

Parameters

- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type

Points

class torchphysics.problem.samplers.sampler_base.**ConcatSampler**(*sampler_a*, *sampler_b*)

Bases: *PointSampler*

A sampler that adds two single samplers together. Will concatenate the data points of both samplers.

Parameters

- **sampler_a** (PointSampler) – The two PointSamplers that should be connected.
- **sampler_b** (PointSampler) – The two PointSamplers that should be connected.

__len__()

Returns the number of points that the sampler will create or has created.

Note: This can be only called if the number of points is set with `n_points`. Elsewise the the number can only be known after the first call to `sample_points` methode or may even change after each call. If you know the number of points yourself, you can set this with `.set_length`.

---

sample_points(*params=Points: {}*, *device='cpu'*)

The method that creates the points. Also implemented in all child classes.

Parameters

- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type

Points

class torchphysics.problem.samplers.sampler_base.**EmptySampler**

Bases: *PointSampler*

A sampler that creates only empty Points. Can be used as a placeholder.

sample_points(*params=Points: {}*, *device='cpu'*, ***kwargs*)

The method that creates the points. Also implemented in all child classes.

Parameters

- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns
A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type
Points

class torchphysics.problem.samplers.sampler_base.**PointSampler**(*n_points=None*, *density=None*, *filter_fn=None*)

Bases: object

Handles the creation and interconnection of training/validation points.

Parameters

- **n_points** (*int, optional*) – The number of points that should be sampled.
- **density** (*float, optional*) – The desired density of the created points.
- **filter_fn** (*callable, optional*) – A function that restricts the possible positions of sample points. A point that is allowed should return True, therefore a point that should be removed must return false. The filter has to be able to work with a batch of inputs. The Sampler will use a rejection sampling to find the right amount of points.

__add__(*other*)

Creates a sampler which samples from two different samples and concatenates both outputs, see `ConcatSampler`.

__iter__()

Creates a iterator of this Pointsampler, with *next* the `sample_points` methode can be called.

__len__()

Returns the number of points that the sampler will create or has created.

Note: This can be only called if the number of points is set with `n_points`. Elsewise the the number can only be known after the first call to `sample_points` methode or may even change after each call. If you know the number of points yourself, you can set this with `.set_length`.

__mul__(*other*)

Creates a sampler that samples from the 'Cartesian product' of the samples of two samplers, see `ProductSampler`.

append(*other*)

Creates a sampler which samples from two different samples and makes a column stack of both outputs, see `AppendSampler`.

classmethod empty(***kwargs*)

Creates an empty Sampler object that samples empty points.

Returns
The empty sampler-object.

Return type
EmptySampler

property is_adaptive

Checks if the Sampler is a AdaptiveSampler, e.g. samples points depending on the loss of the previous iteration.

property is_static

Checks if the Sampler is a StaticSampler, e.g. retuns always the same points.

make_static()

Transforms a sampler to an StaticSampler. A StaticSampler only creates points the first time .sample_points() is called. Afterwards the points are saved and will always be returned if .sample_points() is called again. Useful if the same points should be used while training/validation or if it is not practicall to create new points in each iteration (e.g. grid points).

sample_points(*params=Points: {}*, *device='cpu'*)

The method that creates the points. Also implemented in all child classes.

Parameters

- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type

Points

set_length(*length*)

If a density is used, the number of points will not be known before hand. If len(PointSampler) is needed one can set the expected number of points here.

Parameters

length (*int*) – The expected number of points that this sampler will create.

Notes

If the domain is independent of other variables and a density is used, the sampler will, after the first call to 'sample_points', set this value itself.

class torchphysics.problem.samplers.sampler_base.**ProductSampler**(*sampler_a*, *sampler_b*)

Bases: *PointSampler*

A sampler that constructs the product of two samplers. Will create a meshgrid (Cartesian product) of the data points of both samplers.

Parameters

- **sampler_a** (PointSampler) – The two PointSamplers that should be connected.
- **sampler_b** (PointSampler) – The two PointSamplers that should be connected.

__len__()

Returns the number of points that the sampler will create or has created.

Note: This can be only called if the number of points is set with `n_points`. Elsewise the the number can only be known after the first call to `sample_points` methode or may even change after each call. If you know the number of points yourself, you can set this with `.set_length`.

`sample_points`(*params=Points: {}*, *device='cpu'*)

The method that creates the points. Also implemented in all child classes.

Parameters

- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type

Points

class torchphysics.problem.samplers.sampler_base.**StaticSampler**(*sampler*)

Bases: *PointSampler*

Constructs a sampler that saves the first points created and afterwards always returns these points again. Has the advantage that the points only have to be computed once.

Parameters

sampler (*Pointsampler*) – The basic sampler that will create the points.

`__len__`()

Returns the number of points that the sampler will create or has created.

Note: This can be only called if the number of points is set with `n_points`. Elsewise the the number can only be known after the first call to `sample_points` methode or may even change after each call. If you know the number of points yourself, you can set this with `.set_length`.

`make_static`()

Transforms a sampler to an `StaticSampler`. A StaticSampler only creates points the first time .sample_points() is called. Afterwards the points are saved and will always be returned if .sample_points() is called again. Useful if the same points should be used while training/validation or if it is not practicall to create new points in each iteration (e.g. grid points).

`sample_points`(*params=Points: {}*, *device='cpu'*, ***kwargs*)

The method that creates the points. Also implemented in all child classes.

Parameters

- **params** (*torchphysics.spaces.Points*) – Additional parameters for the domain.
- **device** (*str*) – The device on which the points should be created. Default is 'cpu'.

Returns

A Points-Object containing the created points and, if parameters were passed as an input, the parameters. Whereby the input parameters will get repeated, so that each row of the tensor corresponds to valid point in the given (product) domain.

Return type
Points

2.5 torchphysics package

2.5.1 Subpackages

torchphysics.problem package

Classes and a wrapper to handle the whole problem of a pde, including the pde, variables and their domains and boudary conditions

Subpackages

torchphysics.problem.spaces package

Contains the Spaces and Points classes.

Spaces
It's purpose is to define variable names and dimensions that can be used in functions, domains, models and more.

Points
The `Points` object is a central part of TorchPhysics. They consit of a PyTorch-Tensor and a space. `Points` store data in a tensor with 2-axis, the first corresponding the batch-dimension in a batch of multiple points. The second axis collects the space dimensionalities.

Submodules

torchphysics.problem.spaces.functionspace module

`class torchphysics.problem.spaces.functionspace.FunctionSpace(input_domain, output_space)`

Bases: `object`

A FunctionSpace collects functions that map from a specific input domain to a previously defined output space.

Parameters

- **input_domain** (*torchphysics.Domain*) – The input domain of the functions in this function space.
- **output_space** (*torchphysics.Space*) – The space of the image of the functions in this function space.

torchphysics.problem.spaces.points module

Contains a class that handles the storage of all created data points.

class torchphysics.problem.spaces.points.**Points**(*data*, *space*, ***kwargs*)

Bases: object

A set of points in a space, stored as a torch.Tensor. Can contain multiple axis which keep batch-dimensions.

Parameters

- **data** (*torch.tensor, np.array or list*) – The data points that should be stored. Have to be of the shape (batch_length, space.dimension).
- **space** (*torchphysics.spaces.Space*) – The space to which these points belongs to.

Notes

This class is essentially a combination of a torch.Tensor and a dictionary. So all data points can be stored as a single tensor, where we efficently can access and transform the data. But at the same time have the knowledge of what points belong to which space/variable.

__add__(*other*)

Adds the data of two Points, have to lay in the same space.

__eq__(*other*)

Compares two Points if they are equal.

__getitem__(*val*)

Supports usual slice operations like points[1:3,('x','t')]. Returns a new, sliced, points object.

__iter__()

Iterates through first batch-dim. It is in general not recommended to use this operation because it may lead to huge (and therefore slow) loops.

__len__()

Returns the number of points in this object.

__mul__(*other*)

Pointwise multiplies the data of two Points, have to lay in the same space.

__or__(*other*)

Appends the data points of the second Points behind the data of the first Points in the first batch-dim. (torch.cat((data_1, data_2), dim=0))

__pow__(*other*)

Pointwise raises the data of the first Points object to the power of the second one.

__sub__(*other*)

Substracts the data of two Points, have to lay in the same space.

__truediv__(*other*)

Pointwise divides the data of two Points, have to lay in the same space.

property as_tensor

Returns the underlying tensor.

property coordinates

Returns a dict containing the coordinates of all points for each variable, e.g. {'x': torch.Tensor, 't': torch.Tensor}

cuda(**args*, ***kwargs*)

property device

Returns the device of the underlying tensor.

property dim

Returns the dimension of the points.

classmethod empty(***kwargs*)

Creates an empty Points object.

Returns
The empty Points-object.

Return type
Points

classmethod from_coordinates(*coords*)

Concatenates sample coordinates from a dict to create a point object.

Parameters
coords (*dict*) – The dictionary containing the data for every variable.

Returns
points – the created points object.

Return type
Points

property isempty

Checks whether no points and no structure are saved in this object.

join(*other*)

Stacks the data points of the second Point behind the data of the first Point. (torch.cat((data_1, data_2), dim=-1))

classmethod joined(**points_l*)

Concatenates different Points to one single Points-Object. Will we use torch.cat on the data of the different Points and create the product space of the Points spaces.

Parameters
***points_l** – The different Points that should be connected.

Returns
the created Points object.

Return type
Points

repeat(**n*)

Repeats this points data along the first batch-dimension. Uses torch.repeat and will therefore repeat the data 'batchwise'.

Parameters
n – The number of repeats.

property requires_grad

Returns the '.requires_grad' property of the underlying Tensor.

property shape

The shape of the batch-dimensions of this points object.

to(**args*, ***kwargs*)

Moves the underlying Tensor to other hardware parts.

track_coord_gradients()

unsqueeze(*dim*)

Adds an additional dimension inside the batch dimensions.

Parameters
dim – Where to add the additional axis (considered only inside batch dimensions).

property variables

Returns variables of the points as an unordered set, e.g {'x', 't'}.

torchphysics.problem.spaces.space module

class torchphysics.problem.spaces.space.**R1**(*variable_name*)

Bases: *Space*

The space for one dimensional real numbers.

Parameters
variable_name (*str*) – The name of the variable that belongs to this space.

class torchphysics.problem.spaces.space.**R2**(*variable_name*)

Bases: *Space*

The space for two dimensional real numbers.

Parameters
variable_name (*str*) – The name of the variable that belongs to this space.

class torchphysics.problem.spaces.space.**R3**(*variable_name*)

Bases: *Space*

The space for three dimensional real numbers.

Parameters
variable_name (*str*) – The name of the variable that belongs to this space.

class torchphysics.problem.spaces.space.**Space**(*variables_dims*)

Bases: Counter, OrderedDict

A Space defines (and assigns) the dimensions of the variables that appear in the differentialequation. This class sholud not be instanced directly, rather the corresponding child classes.

Parameters
variables_dims (*dict*) – A dictionary containing the name of the variables and the dimension of the respective variable.

__contains__(*space*)

Checks if the variables of the other space are contained in this space.

Parameters
space (*torchphysics.spaces.Space*) – The other Space that should be checked if this is included.

__getitem__(*val*)

Returns a part of the Space dicitionary, specified in the input. Mathematically, this constructs a subspace.

Parameters

val (*str, slice, list or tuple*) – The keys that correspond to the variables that should be used in the subspace.

__mul__(*other*)

Creates the product space of the two input spaces. Allows the construction of higher dimensional spaces with 'mixed' variable names. E.g R1('x')*R1('y') is a two dimensional space where one axis is 'x' and the other stands for 'y'.

property dim

Returns the dimension of the space (sum of factor spaces)

property variables

A unordered (!) set of variables.

torchphysics.utils package

Useful helper methods for the definition and evaluation of a problem.

For the creation of conditions, some differential operators are implemented under torchphysics.utils.differentialoperators.

For the evaluation of the trained model, some plot and animation functionalities are provided. They can give you a rough overview of the determined solution. These lay under torchphysics.utils.plotting

Subpackages

torchphysics.utils.data package

Submodules

torchphysics.utils.data.dataloader module

class torchphysics.utils.data.dataloader.**DeepONetDataLoader**(*branch_data*, *trunk_data*, *output_data*, *input_space*, *output_space*, *batch_size*, *shuffle=False*, *num_workers=0*, *pin_memory=False*, *drop_last=False*)

Bases: DataLoader

A DataLoader that can be used in a condition to load minibatches of paired data points as the input and output of a DeepONet-model.

Parameters

- **branch_data** (*torch.tensor*) – A tensor containing the input data for the branch network. Shape of the data should be: [number_of_functions, input_dim_of_branch_net]
- **trunk_data** (*torch.tensor*) – A tensor containing the input data for the trunk network. Shape of the data should be: [number_of_functions, number_of_discrete_points, input_dim_of_trunk_net] For each input of the branch_data we will have multiple inputs for the trunk net.

- **output_data** (*torch.tensor*) – A tensor containing the expected output of the network. Shape of the data should be: [number_of_functions, number_of_discrete_points, output_dim].
- **input_space** (*torchphysics.spaces.Space*) – The input space of the trunk network.
- **output_space** (*torchphysics.spaces.Space*) – The output space in which the solution is.
- **batch_size** (*int*) – The size of the loaded batches.
- **shuffle** (*bool*) – Whether to shuffle the order of the data points at initialization.
- **num_workers** (*int*) – The amount of workers used during data loading, see also: the PyTorch documentation
- **pin_memory** (*bool*) – Whether to use pinned memory during data loading, see also: the PyTorch documentation
- **drop_last** (*bool*) – Whether to drop the last (and non-batch-size-) minibatch.

batch_size: Optional[int]

dataset: Dataset[T_co]

drop_last: bool

num_workers: int

pin_memory: bool

prefetch_factor: int

sampler: Union[Sampler, Iterable]

timeout: float

class torchphysics.utils.data.dataloader.**PointsDataLoader**(*data_points*, *batch_size*, *shuffle=False*, *num_workers=0*, *pin_memory=False*, *drop_last=False*)

Bases: DataLoader

A DataLoader that can be used in a condition to load minibatches of paired data points as the input and output of a model.

Parameters

- **data_points** (Points *or* *tuple*) – One or multiple Points object containing multiple data points or tuples of data points. If a tuple of Points objects is given, they should all have the same length, as data will be loaded in tuples where the i-th points are loaded simultaneously.
- **batch_size** (*int*) – The size of the loaded batches.
- **shuffle** (*bool*) – Whether to shuffle the order of the data points at initialization.
- **num_workers** (*int*) – The amount of workers used during data loading, see also: the PyTorch documentation
- **pin_memory** (*bool*) – Whether to use pinned memory during data loading, see also: the PyTorch documentation
- **drop_last** (*bool*) – Whether to drop the last (and non-batch-size-) minibatch.

batch_size: Optional[int]

dataset: Dataset[T_co]

drop_last: bool

num_workers: int

pin_memory: bool

prefetch_factor: int

sampler: Union[Sampler, Iterable]

timeout: float

class torchphysics.utils.data.dataloader.**PointsDataset**(*data_points*, *batch_size*, *shuffle=False*, *drop_last=False*)

Bases: Dataset

A PyTorch Dataset to load tuples of data points.

Parameters

- **data_points** (*Points or tuple*) – One or multiple Points object containing multiple data points or tuples of data points. If a tuple of Points objects is given, they should all have the same length, as data will be loaded in tuples where the i-th points are loaded simultaneously.
- **batch_size** (*int*) – The size of the loaded batches.
- **shuffle** (*bool*) – Whether to shuffle the order of the data points at initialization.
- **drop_last** (*bool*) – Whether to drop the last (and non-batch-size-) minibatch.

__getitem__(*idx*)

Returns the item at the given index.

Parameters

idx (*int*) – The index of the desired point.

__len__()

Returns the number of points of this dataset.

torchphysics.utils.pinn namespace

Submodules

torchphysics.utils.pinn.differentialequations module

class torchphysics.utils.pinn.differentialequations.**BurgersEquation**(*viscosity*, *spatial_var='x'*, *time_var='t'*)

Bases: Module

Implementation of the viscous Burgers equation: $u_t + (u \cdot \nabla) \cdot u - viscosity \Delta u = 0$.

Parameters

- **viscosity** (*scalar or Parameter*) – The viscosity coefficient of the burgers equation. Should be either a scalar constant, or a Parameter that will be learned based on data. If 0, the inviscid Burgers equation will be solved.

- **spatial_var** (*str*) – Name of the spatial variable. Defaults to 'x'.
- **time_var** (*str*) – Name of the time variable. Defaults to 't'.

forward(*u*, ***inputs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class torchphysics.utils.pinn.differentialequations.**HeatEquation**(*diffusivity*, *spatial_var='x'*, *time_var='t'*)

Bases: `Module`

Implementation of the homogenous heat equation: $u_t - D\Delta u = 0$.

Parameters

- **diffusivity** (*scalar or torchphysics.models.Parameter*) – The diffusivity coefficient of the heat equation. Should be either a scalar constant, or a Parameter that will be learned based on data.
- **spatial_var** (*str*) – Name of the spatial variable. Defaults to 'x'.
- **time_var** (*str*) – Name of the time variable. Defaults to 't'.

forward(*u*, ***inputs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class torchphysics.utils.pinn.differentialequations.**IncompressibleNavierStokesEquation**(*viscosity*, *spatial_var='x'*, *time_var='t'*)

Bases: `Module`

Implementation of the incompressible Navier Stokes equation. Needs two outputs of the network: The velocity field and the pressure.

Parameters

- **viscosity** (*scalar or Parameter*) – The viscosity coefficient of the burgers equation. Should be either a scalar constant, or a Parameter that will be learned based on data. If 0, the inviscid Burgers equation will be solved.
- **spatial_var** (*str*) – Name of the spatial variable. Defaults to 'x'.
- **time_var** (*str*) – Name of the time variable. Defaults to 't'.

training: bool

torchphysics.utils.plotting package

Different plotting functions:

- `plot_functions` implement functions to show the output of the neural network or values derivative from it (derivatives, ...).
- `animation` implement the same concepts as the plot functions, just as animations.
- `scatter_points` are meant to show a batch of used training points of a sampler.

Submodules

torchphysics.utils.plotting.animation module

This file contains different functions for animating the output of the neural network

`torchphysics.utils.plotting.animation.`**`animate`**(*model*, *ani_function*, *ani_sampler*, *ani_speed=50*, *angle=[30, 30]*, *ani_type=''*)

Main function for animations.

Parameters

- **model** (*torchphysics.models.Model*) – The Model/neural network that should be used in the plot.
- **ani_function** (*Callable*) – A function that specfices the part of the model that should be animated. Of the same form as the plot function.
- **point_sampler** (*torchphysics.samplers.AnimationSampler*) – A Sampler that creates the points that should be used for the animation.
- **angle** (*list, optional*) – The view angle for 3D plots. Standard angle is [30, 30]
- **ani_type** (*str, optional*) – Specifies how the output sholud be animated. If no input is given, the method will try to use a fitting way, to show the data. Implemented types are:
  - 'line' for line animations, with a 1D-domain and output
  - 'surface_2D' for surface animation, with a 2D-domain
  - 'quiver_2D' for quiver/vector field animation, with a 2D-domain
  - 'contour_surface' for contour/colormaps, with a 2D-domain

Returns

- *plt.figure* – The figure handle of the created plot
- *animation.FuncAnimation* – The function that handles the animation

Notes

This methode only creates a simple animation and is for complex domains not really optimized. Should only be used to get a rough understanding of the trained neural network.

torchphysics.utils.plotting.animation.**animation_contour_2D**(*outputs*, *ani_sampler*, *animation_points*, *domain_points*, *angle*, *ani_speed*)

Handles colormap animations in 2D

torchphysics.utils.plotting.animation.**animation_line**(*outputs*, *ani_sampler*, *animation_points*, *domain_points*, *angle*, *ani_speed*)

Handels 1D animations, inputs are the same as animation().

torchphysics.utils.plotting.animation.**animation_quiver_2D**(*outputs*, *ani_sampler*, *animation_points*, *domain_points*, *angle*, *ani_speed*)

Handles quiver animations in 2D

torchphysics.utils.plotting.animation.**animation_surface2D**(*outputs*, *ani_sampler*, *animation_points*, *domain_points*, *angle*, *ani_speed*)

Handels 2D animations, inputs are the same as animation().

torchphysics.utils.plotting.plot_functions module

This file contains different functions for plotting outputs of neural networks

class torchphysics.utils.plotting.plot_functions.**Plotter**(*plot_function*, *point_sampler*, *angle=[30, 30]*, *log_interval=None*, *plot_type=''*, ***kwargs*)

Bases: `object`

Object to collect plotting properties.

Parameters

- **plot_function** (*callable*) – A function that specfices the part of the model that should be plotted. Can be of the same form as the condition-functions. E.g. if the solution name is 'u' we can use

  plot_func(u):
  return u[:, 0]

  to plot the first entry of 'u'. For the derivative we could write:

  plot_func(u, x):
  return grad(u, x)

- **point_sampler** (*torchphysics.samplers.PlotSampler*) – A Sampler that creates the points that should be used for the plot.
- **angle** (*list, optional*) – The view angle for surface plots. Standart angle is [30, 30]

- **log_interval** (*int*) – Plots will be saved every log_interval steps if the plotter is used in training of a model.
- **plot_type** (*str, optional*) – Specifies how the output should be plotted. If no input is given, the method will try to use a fitting way, to show the data. Implemented types are:
  - 'line' for plots in 1D
  - 'surface_2D' for surface plots, with a 2D-domain
  - 'curve' for a curve in 3D, with a 1D-domain,
  - 'quiver_2D' for quiver/vector field plots, with a 2D-domain
  - 'contour_surface' for contour/colormaps, with a 2D-domain
- **kwargs** – Additional arguments to specify different parameters/behaviour of the plot. See https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html for possible arguments of each underlying object.

plot(*model*)

Creates the plot of the model.

Parameters

model (*torchphysics.models.Model*) – The Model/neural network that should be used in the plot.

Returns

The figure handle of the created plot

Return type

plt.figure

torchphysics.utils.plotting.plot_functions.**contour_2D**(*output*, *domain_points*, *point_sampler*, *angle*, ***kwargs*)

Handles colormap/contour plots w.r.t. a two dimensional variable.

torchphysics.utils.plotting.plot_functions.**curve3D**(*output*, *domain_points*, *point_sampler*, *angle*, ***kwargs*)

Handles curve plots where the output is 2D and the domain is 1D.

torchphysics.utils.plotting.plot_functions.**line_plot**(*output*, *domain_points*, *point_sampler*, *angle*, ***kwargs*)

Handels line plots w.r.t. a one dimensional variable.

torchphysics.utils.plotting.plot_functions.**plot**(*model*, *plot_function*, *point_sampler*, *angle=[30, 30]*, *plot_type=''*, *device='cpu'*, ***kwargs*)

Main function for plotting

Parameters

- **model** (*torchphysics.models.Model*) – The Model/neural network that should be used in the plot.
- **plot_function** (*callable*) – A function that specfices the part of the model that should be plotted. Of the same form as the condition-functions. E.g. if the solution name is 'u', we can use

```
plot_func(u):
    return u[:, 0]
```

to plot the first entry of 'u'. For the derivative we could write:

```
plot_func(u, x):
    return grad(u, x)
```

- **point_sampler** (*torchphysics.samplers.PlotSampler*) – A Sampler that creates the points that should be used for the plot.
- **angle** (*list, optional*) – The view angle for 3D plots. Standard angle is [30, 30]
- **plot_type** (*str, optional*) – Specifies how the output sholud be plotted. If no input is given the method will try to use a fitting way to show the data. Implemented types are:
  - 'line' for plots in 1D
  - 'surface_2D' for surface plots, with a 2D-domain
  - 'curve' for a curve in 3D, with a 1D-domain,
  - 'quiver_2D' for quiver/vector-field plots, with a 2D-domain
  - 'contour_surface' for contour/colormaps, with a 2D-domain
- **kwargs** – Additional arguments to specify different parameters/behaviour of the plot. See https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html for possible arguments of each underlying object.

Returns
The figure handle of the created plot

Return type
plt.figure

Notes

What this function does is: creating points with sampler -> evaluate model -> evalute plot function -> create the plot with matplotlib.pyplot.

The function is only meant to give a fast overview over the trained neural network. In general the methode is not optimized for complex domains.

torchphysics.utils.plotting.plot_functions.**quiver2D**(*output*, *domain_points*, *point_sampler*, *angle*, ***kwargs*)

Handles quiver/vector field plots w.r.t. a two dimensional variable.

torchphysics.utils.plotting.plot_functions.**surface2D**(*output*, *domain_points*, *point_sampler*, *angle*, ***kwargs*)

Handels surface plots w.r.t. a two dimensional variable.

torchphysics.utils.plotting.scatter_points module

Function to show an example of the created points of the sampler.

torchphysics.utils.plotting.scatter_points.**scatter**(*subspace*, **samplers*)

Shows (one batch) of used points in the training. If the sampler is static, the shown points will be the points for the training. If not the points may vary, depending of the sampler.

Parameters

- **subspace** (*torchphysics.problem.Space*) – The (sub-)space of which the points should be plotted. Only plotting for dimensions <= 3 is possible.
- ***samplers** (*torchphysics.problem.Samplers*) – The diffrent samplers for which the points should be plotted. The plot for each sampler will be created in the order there were passed in.

Returns

fig – The figure handle of the plot.

Return type

matplotlib.pyplot.figure

Submodules

torchphysics.utils.callbacks module

class torchphysics.utils.callbacks.**PlotterCallback**(*model*, *plot_function*, *point_sampler*, *log_name='plot'*, *check_interval=200*, *angle=[30, 30]*, *plot_type=''*, ***kwargs*)

Bases: Callback

Object for plotting (logging plots) inside of tensorboard. Can be passed to the pytorch lightning trainer.

Parameters

- **plot_function** (*callable*) – A function that specfices the part of the model that should be plotted.
- **point_sampler** (*torchphysics.samplers.PlotSampler*) – A sampler that creates the points that should be used for the plot.
- **log_interval** (*str, optional*) – Name of the plots inside of tensorboard.
- **check_interval** (*int, optional*) – Plots will be saved every check_interval steps, if the plotter is used.
- **angle** (*list, optional*) – The view angle for surface plots. Standard angle is [30, 30]
- **plot_type** (*str, optional*) – Specifies how the output should be plotted. If no input is given, the method will try to use a fitting way, to show the data. See also plot-functions.
- **kwargs** – Additional arguments to specify different parameters/behaviour of the plot. See https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html for possible arguments of each underlying object.

on_train_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)

Called when the train batch ends.

on_train_end(*trainer*, *pl_module*)

Called when the train ends.

on_train_start(*trainer*, *pl_module*)

Called when the train begins.

class torchphysics.utils.callbacks.**WeightSaveCallback**(*model*, *path*, *name*, *check_interval*, *save_initial_model=False*, *save_final_model=True*)

Bases: `Callback`

A callback to save the weights of a model during training. Can save the model weights before, during and after training. During training, only the model with minimal loss will be saved.

Parameters

- **model** (*torch.nn.Module*) – The model of which the weights should be saved.
- **path** (*str*) – The relative path of the saved weights.
- **name** (*str*) – A name that will become part of the file name of the saved weights.
- **check_interval** (*int*) – The callback will check for minimal loss every check_interval iterations. If negative, no weights will be saved during training.
- **save_initial_model** (*False*) – Whether the model should be saved before training as well.
- **save_final_model** (*True*) – Whether the model should always be saved after the last iteration.

on_train_batch_start(*trainer*, *pl_module*, *batch*, *batch_idx*, *dataloader_idx*)

Called when the train batch begins.

on_train_end(*trainer*, *pl_module*)

Called when the train ends.

on_train_start(*trainer*, *pl_module*)

Called when the train begins.

torchphysics.utils.differentialoperators module

File contains differentialoperators

torchphysics.utils.differentialoperators.**convective**(*deriv_out*, *convective_field*, **derivative_variable*)

Computes the convective term $(v \cdot \nabla)u$ that appears e.g. in material derivatives. Note: This is not the whole material derivative.

Parameters

- **deriv_out** (*torch.tensor*) – The vector or scalar field u that is convected and should be differentiated.
- **convective_field** (*torch.tensor*) – The flow vector field v. Should have the same dimension as derivative_variable.
- **derivative_variable** (*torch.tensor*) – The spatial variable in which respect deriv_out should be differentiated.

Returns

A vector or scalar (+batch-dimension) Tensor, that contains the convective derivative.

Return type
torch.tensor

torchphysics.utils.differentialoperators.**div**(*model_out*, **derivative_variable*)

Computes the divergence of a network with respect to the given variable. Only for vector valued inputs, for matices use the function matrix_div.

Parameters

- **model_out** (*torch.tensor*) – The output tensor of the neural network
- **derivative_variable** (*torch.tensor*) – The input tensor of the variables in which respect the derivatives have to be computed. Have to be in a consistent ordering, if for example the output is u = (u_x, u_y) than the variables has to passed in the order (x, y)

Returns
A Tensor, where every row contains the values of the divergence of the model w.r.t the row of the input variable.

Return type
torch.tensor

torchphysics.utils.differentialoperators.**grad**(*model_out*, **derivative_variable*)

Computes the gradient of a network with respect to the given variable.

Parameters

- **model_out** (*torch.tensor*) – The (scalar) output tensor of the neural network
- **derivative_variable** (*torch.tensor*) – The input tensor of the variables in which respect the derivatives have to be computed

Returns
A Tensor, where every row contains the values of the the first derivatives (gradient) w.r.t the row of the input variable.

Return type
torch.tensor

torchphysics.utils.differentialoperators.**jac**(*model_out*, **derivative_variable*)

Computes the jacobian of a network output with respect to the given input.

Parameters

- **model_out** (*torch.tensor*) – The output tensor in which respect the jacobian should be computed.
- **derivative_variable** (*torch.tensor*) – The input tensor in which respect the jacobian should be computed.

Returns
A Tensor of shape (b, m, n), where every row contains a jacobian.

Return type
torch.tensor

torchphysics.utils.differentialoperators.**laplacian**(*model_out*, **derivative_variable*, *grad=None*)

Computes the laplacian of a network with respect to the given variable

Parameters

- **model_out** (*torch.tensor*) – The (scalar) output tensor of the neural network
- **derivative_variable** (*torch.tensor*) – The input tensor of the variables in which respect the derivatives have to be computed

- **grad** (*torch.tensor*) – If the gradient has already been computed somewhere else, it is more efficient to use it again.

Returns
A Tensor, where every row contains the value of the sum of the second derivatives (laplace) w.r.t the row of the input variable.

Return type
torch.tensor

torchphysics.utils.differentialoperators.**matrix_div**(*model_out*, **derivative_variable*)

Computes the divergence for matrix/tensor-valued functions.

Parameters

- **model_out** (*torch.tensor*) – The (batch) of matirces that should be differentiated.
- **derivative_variable** (*torch.tensor*) – The spatial variable in which respect should be differentiated.

Returns
A Tensor of vectors of the form (batch, dim), containing the divegrence of the input.

Return type
torch.tensor

torchphysics.utils.differentialoperators.**normal_derivative**(*model_out*, *normals*, **derivative_variable*)

Computes the normal derivativ of a network with respect to the given variable and normal vectors.

Parameters

- **model_out** (*torch.tensor*) – The (scalar) output tensor of the neural network
- **derivative_variable** (*torch.tensor*) – The input tensor of the variables in which respect the derivatives have to be computed
- **normals** (*torch.tensor*) – The normal vectors at the points where the derivative has to be computed. In the form: normals = tensor([normal_1, normal_2, . . .]

Returns
A Tensor, where every row contains the values of the normal derivatives w.r.t the row of the input variable.

Return type
torch.tensor

torchphysics.utils.differentialoperators.**partial**(*model_out*, **derivative_variables*)

Computes the (n-th, possibly mixed) partial derivative of a network output with respect to the given variables.

Parameters

- **model_out** (*torch.tensor*) – The output tensor of the neural network
- **derivative_variables** (*torch.tensor(s)*) – The input tensors in which respect the derivatives should be computed. If n tensors are given, the n-th (mixed) derivative will be computed.

Returns
A Tensor, where every row contains the values of the computed partial derivative of the model w.r.t the row of the input variable.

Return type
torch.tensor

torchphysics.utils.differentialoperators.**rot**(*model_out*, **derivative_variable*)

Computes the rotation/curl of a 3-dimensional vector field (given by a network output) with respect to the given input.

Parameters

- **model_out** (*torch.tensor*) – The output tensor of shape (b, 3) in which respect the roation should be computed.
- **derivative_variable** (*torch.tensor*) – The input tensor of shape (b, 3) in which respect the rotation should be computed.

Returns

A Tensor of shape (b, 3), where every row contains a rotation/curl vector for a given batch element.

Return type

torch.tensor

torchphysics.utils.differentialoperators.**sym_grad**(*model_out*, **derivative_variable*)

Computes the symmetric gradient: $0.5(ablau + ablau^T)$.

model_out

[torch.tensor] The vector field u that should be differentiated.

derivative_variable

[torch.tensor] The spatial variable in which respect model_out should be differentiated.

torch.tensor

A Tensor of matrices of the form (batch, dim, dim), containing the symmetric gradient.

torchphysics.utils.evaluation module

File contains different helper functions to get specific informations about the computed solution.

torchphysics.utils.evaluation.**compute_min_and_max**(*model*, *sampler*, *evaluation_fn=<function <lambda>>*, *device='cpu'*, *requieres_grad=False*)

Computes the minimum and maximum values of the model w.r.t. the given variables.

Parameters

- **model** (*DiffEqModel*) – A neural network of which values should be computed.
- **sampler** (*torchphysics.samplers.PointSampler*) – A sampler that creates the points where the model should be evaluated.
- **evaluation_fn** (*callable*) – A user-defined function that uses the neural network and creates the desiered output quantity.
- **device** (*str or torch device*) – The device of the model.
- **track_gradients** (*bool*) – Whether to track input gradients or not.

Returns

- *float* – The minimum value computed.
- *float* – The maximum value computed.

torchphysics.utils.user_fun module

Contains a class which extracts the needed arguments of an arbitrary methode/function and wraps them for future usage. E.g correctly choosing the needed arguments and passing them on to the original function.

class torchphysics.utils.user_fun.**DomainUserFunction**(*fun*, *defaults={}*, *args={}*)

Bases: *UserFunction*

Extension of the original UserFunctions, that are used in the Domain-Class.

Parameters

- **fun** (*callable*) – The original function that should be wrapped.
- **defaults** (*dict, optional*) – Possible defaults arguments of the function. If none are specified will check by itself if there are any.
- **args** (*dict, optional*) – All arguments of the function. If none are specified will check by itself if there are any.

Notes

The only difference to normal UserFunction is how the evaluation of the original function is handled. Since all Domains use Pytorch, we check that the output always is a torch.tensor. In the case that the function is not constant, we also append an extra dimension to the output, so that the domains can work with it correctly.

__call__(*args={}*, *device='cpu'*)

To evalute the function. Will automatically extract the needed arguments from the input data and will set the possible default values.

Parameters

- **args** (*dict or torchphysics.Points*) – The input data, where the function should be evaluated.
- **device** (*str, optional*) – The device on which the output of th efunction values should lay. Default is 'cpu'.

Returns

The output values of the function.

Return type

torch.tensor

evaluate_function(*device='cpu'*, ***inp*)

Evaluates the original input function. Should not be used directly, rather use the call-methode.

Parameters

- **device** (*str, optional*) – The device on which the output of th efunction values should lay. Default is 'cpu'.
- **inp** – The input values.

class torchphysics.utils.user_fun.**UserFunction**(*fun*, *defaults={}*, *args={}*)

Bases: object

Wraps a function, so that it can be called with arbitrary input arguments.

Parameters

- **fun** (*callable*) – The original function that should be wrapped.

- **defaults** (*dict, optional*) – Possible defaults arguments of the function. If none are specified will check by itself if there are any.
- **args** (*dict, optional*) – All arguments of the function. If none are specified will check by itself if there are any.

Notes

Uses inspect.getfullargspec(fun) to get the possible input arguments. When called just extracts the needed arguments and passes them to the original function.

__call__(*args={}*, *vectorize=False*)

To evalute the function. Will automatically extract the needed arguments from the input data and will set the possible default values.

Parameters

- **args** (*dict or torchphysics.Points*) – The input data, where the function should be evaluated.
- **vectorize** (*bool, optional*) – If the original function can work with a batch of data, or a loop needs to be used to evaluate the function. default is False, which means that we assume the function can work with a batch of data.

Returns
The output values of the function.

Return type
torch.tensor

apply_to_batch(*inp*)

Apply the function to a batch of elements by running a for-loop. we assume that all inputs either have batch (i.e. maximum) dimension or are a constant param.

Parameters
inp (*torchphysics.points*) – The Points-object of the input data

Returns
The output values of the function, for each input.

Return type
torch.tensor

evaluate_function(***inp*)

Evaluates the original input function. Should not be used directly, rather use the call-methode.

property necessary_args

Returns the function arguments that are needed to evaluate this function.

Returns
The needed arguments.

Return type
list

property optional_args

Returns the function arguments that are optional to evaluate this function.

Returns
The optional arguments.

Return type
list

`partially_evaluate`(***args*)

(partially) evaluates a given function.

Parameters
****args** – The arguments where the function should be (partially) evaluated.

Returns
Out – If the input arguments are enough to evalate the whole function, the corresponding output is returned. If some needed arguments are missing, a copy of this UserFunction will be returned. Whereby the values of **args will be added to the default values of the returned UserFunction.

Return type
value or *UserFunction*

`remove_default`(**args*, ***kwargs*)

Removes an default value of a input argument.

Parameters

- ***args** – The arguments for which the default values should be deleted.
- ****kwargs** – The arguments for which the default values should be deleted.

`set_default`(***args*)

Sets a input argument to given value.

Parameters
****args** – The value the input should be set to.

2.5.2 Submodules

2.5.3 torchphysics.solver module

`class torchphysics.solver.OptimizerSetting`(*optimizer_class*, *lr*, *optimizer_args={}*, *scheduler_class=None*, *scheduler_args={}*, *scheduler_frequency=1*)

Bases: `object`

A helper class to sum up the optimization setup in a single class.

`class torchphysics.solver.Solver`(*train_conditions*, *val_conditions=()*, *optimizer_setting=<torchphysics.solver.OptimizerSetting object>*)

Bases: `LightningModule`

A LightningModule that handles optimization and metric logging of given conditions.

Parameters

- **train_conditions** (*tuple or list*) – Tuple or list of conditions to be optimized. The weighted sum of their losses will be computed and minimized.
- **val_conditions** (*tuple or list*) – Conditions to be tracked during the validation part of the training, can be used e.g. to track errors comparede to measured data.
- **optimizer_setting** (`OptimizerSetting`) – A OptimizerSetting object that contains all necessary parameters for optimizing, see *OptimizerSetting*.

`configure_optimizers()`

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
```

(continues on next page)

(continued from previous page)

```
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, *training_step()* will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current opti-

mizer at each training step.

- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

on_train_start()

Called at the beginning of training after sanity check.

train_dataloader()

Implement one or more PyTorch DataLoaders for training.

Returns

A collection of `torch.utils.data.DataLoader` specifying training samples. In the case of multiple dataloaders, please see this section.

The dataloader you return will not be reloaded unless you set **:paramref:`~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- `prepare_data()`
- `setup()`

Note: Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Example:

```
# single dataloader
def train_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=True, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=True
    )
    return loader

# multiple dataloaders, return as list
def train_dataloader(self):
```

(continues on next page)

(continued from previous page)

```
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a list of tensors: [batch_mnist, batch_cifar]
    return [mnist_loader, cifar_loader]

# multiple dataloader, return as dict
def train_dataloader(self):
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a dict of tensors: {'mnist': batch_mnist, 'cifar': batch_
↪cifar}
    return {'mnist': mnist_loader, 'cifar': cifar_loader}
```

training: bool

training_step(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (int) – Integer displaying index of this batch
- **optimizer_idx** (int) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Any) – Passed in if **:paramref:`~pytorch_lightning.core.lightning.LightningModule.truncate** > 0.

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- **None - Training will skip to the next batch. This is only for automatic optimization.**
  This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

`val_dataloader()`

Implement one or multiple PyTorch DataLoaders for validation.

The dataloader you return will not be reloaded unless you set **:paramref:`~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `fit()`
- `validate()`
- `prepare_data()`
- `setup()`

Note: Lightning adds the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Returns

A `torch.utils.data.DataLoader` or a sequence of them specifying validation samples.

Examples:

```
def val_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False,
                    transform=transform, download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def val_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

Note: If you don't need a validation dataset and a *validation_step()*, you don't need to implement this method.

Note: In the case where you return multiple validation dataloaders, the *validation_step()* will have an argument `dataloader_idx` which matches the order here.

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the *validation_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

CHAPTER

THREE

BIBLIOGRAPHY

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

N

Q

R

S

T

U

V

W