idrlnet

Release 0.0.2-rc3

IDRL

Jun 29, 2023

CONTENTS

CHAPTER

ONE

INSTALLATION

We recommend using conda to manage the environment. Other methods may also work well such like using docker or virtual env.

Choose one of the following installation methods.

1.1 PyPI

Simple installation from PyPI

```
pip install -U idrlnet
```

Note: To avoid version conflicts, please use some tools to create a virtual environment first.

1.2 Docker

Pull latest docker image from Dockerhub.

```
docker pull idrl/idrlnet:latest
docker run -it idrl/idrlnet:latest bash
```

Note: Available tags can be found in Dockerhub.

1.3 Anaconda

```
conda create -n idrlnet_dev python=3.8 -y
conda activate idrlnet_dev
pip install idrlnet
```

1.4 From Source

```
git clone https://github.com/idrl-lab/idrlnet
cd idrlnet
pip install -e .
```

CHAPTER

TWO

TUTORIAL

To make full use of IDRLnet. We strongly suggest following the following examples:

1. *Simple Poisson*. This example introduces the primary usage of IDRLnet. Including creating sampling domains, neural networks, partial differential equations, training, monitoring, and inference.
2. *Euler-Bernoulli beam*. The example introduces how to use symbols to construct a PDE node efficiently.
3. *Burgers' Equation*. The case presents how to include `time` in the sampling domains.
4. *Allen-Cahn Equation*. The example introduces the representation of periodic boundary conditions. `Receiver` acting as `callbacks` are also introduced, including implementing user-defined algorithms and post-processing during the training.
5. *Inverse wave equation*. The example introduces how to discover unknown parameters in PDEs.
6. *Parameterized poisson equation*. The example introduces how to train a surrogate with parameters.
7. *Variational Minimization*. The example introduces how to solve variational minimization problems.
8. *Volterra integral differential equation*. The example introduces the way to solve IDEs.
9. *Navier-Stokes equation*. The example introduces how to use the LBFGS optimizer.
10. *Deepritz method*. The example introduces the way to solve PDEs with the Deepritz method.

2.1 Solving Simple Poisson Equation

Inspired by Nvidia SimNet, IDRLnet employs symbolic links to construct a computational graph automatically. In this section, we introduce the primary usage of IDRLnet. To solve PINN via IDRLnet, we divide the procedure into several parts:

1. Define symbols and parameters.
2. Define geometry objects.
3. Define sampling domains and corresponding constraints.
4. Define neural networks and PDEs.
5. Define solver and solve.
6. Post processing.

We provide the following example to illustrate the primary usages and features of IDRLnet.

Consider the 2d Poisson's equation defined on $\Omega=[-1,1]\times[-1,1]$, which satisfies $-\Delta u=1$, with the boundary value conditions:

$$ \begin{align} \frac{\partial u(x, -1)}{\partial n}&=\frac{\partial u(x, 1)}{\partial n}=0 \ u(-1,y)&=u(1, y)=0 \end{align} $$

2.1.1 Define Symbols

For the 2d problem, we define two coordinate symbols `x` and `y`, which will be used in symbolic expressions in IDRLnet.

```
x, y = sp.symbols('x y')
```

Note that variables `x`, `y`, `z`, `t` are reserved inside IDRLnet. The four symbols should only represent the 4 primary coordinates.

2.1.2 Define Geometric Objects

The geometry object is a simple rectangle.

```
rec = sc.Rectangle((-1., -1.), (1., 1.))
```

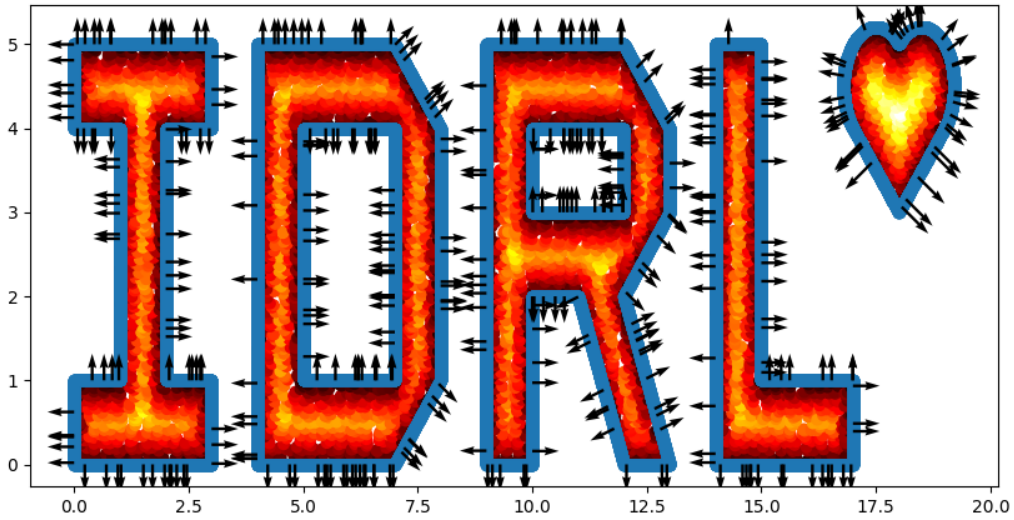
Users can sample points on these geometry objects. The operators +, -, & are also supported. A slightly more complicated example is as follows:

```
import numpy as np
import idrlnet.shortcut as sc

# Define 4 polygons
I = sc.Polygon([(0, 0), (3, 0), (3, 1), (2, 1), (2, 4), (3, 4), (3, 5), (0, 5), (0, 4),␣
↪(1, 4), (1, 1), (0, 1)])
D = sc.Polygon([(4, 0), (7, 0), (8, 1), (8, 4), (7, 5), (4, 5)]) - sc.Polygon(([5, 1],␣
↪[7, 1], [7, 4], [5, 4]))
R = sc.Polygon([(9, 0), (10, 0), (10, 2), (11, 2), (12, 0), (13, 0), (12, 2), (13, 3),␣
↪(13, 4), (12, 5), (9, 5)]) \
    - sc.Rectangle(point_1=(10., 3.), point_2=(12, 4))
L = sc.Polygon([(14, 0), (17, 0), (17, 1), (15, 1), (15, 5), (14, 5)])

# Define a heart shape.
heart = sc.Heart((18, 4), radius=1)

# Union of the 5 geometry objects
geo = (I + D + R + L + heart)

# interior samples
points = geo.sample_interior(density=100, low_discrepancy=True)
plt.figure(figsize=(10, 5))
plt.scatter(x=points['x'], y=points['y'], c=points['sdf'], cmap='hot')

# boundary samples
points = geo.sample_boundary(density=400, low_discrepancy=True)
plt.scatter(x=points['x'], y=points['y'])
idx = np.random.choice(points['x'].shape[0], 400, replace=False)

# Show normal directions on boundary
plt.quiver(points['x'][idx], points['y'][idx], points['normal_x'][idx], points['normal_y
```

(continues on next page)

(continued from previous page)

```
↪'][idx])
plt.show()
```



2.1.3 Define Sampling Methods and Constraints

Take a 1D fitting task as an example. The data source generates pairs (x_i, f_i). We train a network $u_\theta(x_i)\approx f_i$. Then f_i is the target output of $u_\theta(x_i)$. These targets are called constraints in IDRLnet.

For the problem, three constraints are presented.

The constraint

$$ u(-1,y)=u(1, y)=0 $$ is translated into

```
@sc.datanode
class LeftRight(sc.SampleDomain):
    # Due to `name` is not specified, LeftRight will be the name of datanode automatically
    def sampling(self, *args, **kwargs):
        # sieve define rules to filter points
        points = rec.sample_boundary(1000, sieve=((y > -1.) & (y < 1.)))
        constraints = sc.Variables({'T': 0.})
        return points, constraints
```

Then `LeftRight()` is wrapped as an instance of `DataNode`. One can store states in these instances. Alternatively, if users do not need storing states, the code above is equivalent to

```
@sc.datanode(name='LeftRight')
def leftright(self, *args, **kwargs):
    points = rec.sample_boundary(1000, sieve=((y > -1.) & (y < 1.)))
    constraints = sc.Variables({'T': 0.})
    return points, constraints
```

Then `sampling()` is wrapped as an instance of `DataNode`.

The constraint

$$ \frac{\partial u(x, -1)}{\partial n}=\frac{\partial u(x, 1)}{\partial n}=0 $$ is translated into

```
@sc.datanode(name="up_down")
class UpDownBoundaryDomain(sc.SampleDomain):
    def sampling(self, *args, **kwargs):
        points = rec.sample_boundary(1000, sieve=((x > -1.) & (x < 1.)))
        constraints = sc.Variables({'normal_gradient_T': 0.})
        return points, constraints
```

The constraint `normal_gradient_T` will also be one of the output of computable nodes, including `PdeNode` or `NetNode`.

The last constraint is the PDE itself $-\Delta u=1$:

```
@sc.datanode(name="heat_domain")
class HeatDomain(sc.SampleDomain):
    def __init__(self):
        self.points = 1000

    def sampling(self, *args, **kwargs):
        points = rec.sample_interior(self.points)
        constraints = sc.Variables({'diffusion_T': 1.})
        return points, constraints
```

`diffusion_T` will also be one of the outputs of computable nodes. `self.points` is a stored state and can be varied to control the sampling behaviors.

2.1.4 Define Neural Networks and PDEs

As mentioned before, neural networks and PDE expressions are encapsulated as `Node` too. The `Node` objects have `inputs`, `derivatives`, `outputs` properties and the `evaluate()` method. According to their inputs, derivatives, and outputs, these nodes will be automatically connected as a computational graph. A topological sort will be applied to the graph to decide the computation order.

```
net = sc.get_net_node(inputs=('x', 'y',), outputs=('T',), name='net1', arch=sc.Arch.mlp)
```

This is a simple call to get a neural network with the predefined architecture. As an alternative, one can specify the configurations via

```
evaluate = MLP(n_seq=[2, 20, 20, 20, 20, 1)],
               activation=Activation.swish,
               initialization=Initializer.kaiming_uniform,
               weight_norm=True)
net = NetNode(inputs=('x', 'y',), outputs=('T',), net=evaluate, name='net1', *args,␣
↪**kwargs)
```

which generates a node with

- `inputs=('x','y')`,
- `derivatives=tuple()`,
- `outpus=('T')`

```
pde = sc.DiffusionNode(T='T', D=1., Q=0., dim=2, time=False)
```

generates a node with

- `inputs=tuple()`,
- `derivatives=('T__x', 'T__y')`,
- `outputs=('diffusion_T',)`.

```
grad = sc.NormalGradient('T', dim=2, time=False)
```

generates a node with

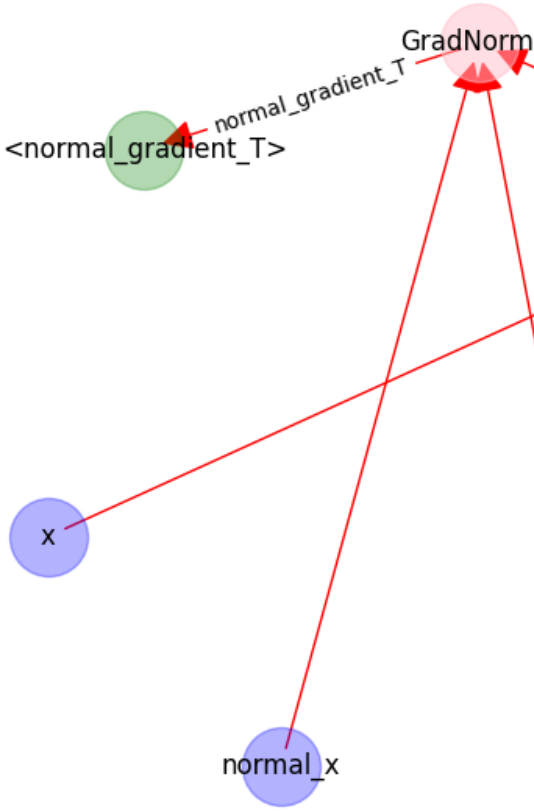
- `inputs=('normal_x', 'normal_y')`,
- `derivatives=('T__x', 'T__y')`,
- `outputs=('normal_gradient_T',)`. The string `__` is reserved to represent the derivative operator. If the required derivatives cannot be directly obtained from outputs of other nodes, It will try `autograd` provided by Pytorch with the maximum prefix match from outputs of other nodes.

2.1.5 Define A Solver

Initialize a solver to bundle all the components and solve the model.

```
s = sc.Solver(sample_domains=(HeatDomain(), LeftRight(), UpDownBoundaryDomain()),
              netnodes=[net],
              pdes=[pde, grad],
              max_iter=1000)
s.solve()
```

Before the solver start running, it constructs computational graphs and applies a topological sort to decide the evaluation order. Each sample domain has its independent graph. The procedures will be executed automatically when the solver detects potential changes in graphs. As default, these graphs are also visualized as `png` in the `network` directory named after the corresponding domain.



The following figure shows the graph on `UpDownBoundaryDomain`:

- The blue nodes are generated via sampling;
- the red nodes are computational;
- the green nodes are constraints(targets).

2.1.6 Inference

We use domain `heat_domain` for inference. First, we increase the density to 10000 via changing the attributes of the domain. Then, `Solver.infer_step()` is called for inference.

```
s.set_domain_parameter('heat_domain', {'points': 10000})
coord = s.infer_step({'heat_domain': ['x', 'y', 'T']})
num_x = coord['heat_domain']['x'].cpu().detach().numpy().ravel()
num_y = coord['heat_domain']['y'].cpu().detach().numpy().ravel()
num_Tp = coord['heat_domain']['T'].cpu().detach().numpy().ravel()
```

One may also define a separate domain for inference, which generates `constraints={}`, and thus, no computational graphs will be generated on the domain. We will see this later.

2.1.7 Performance Issues

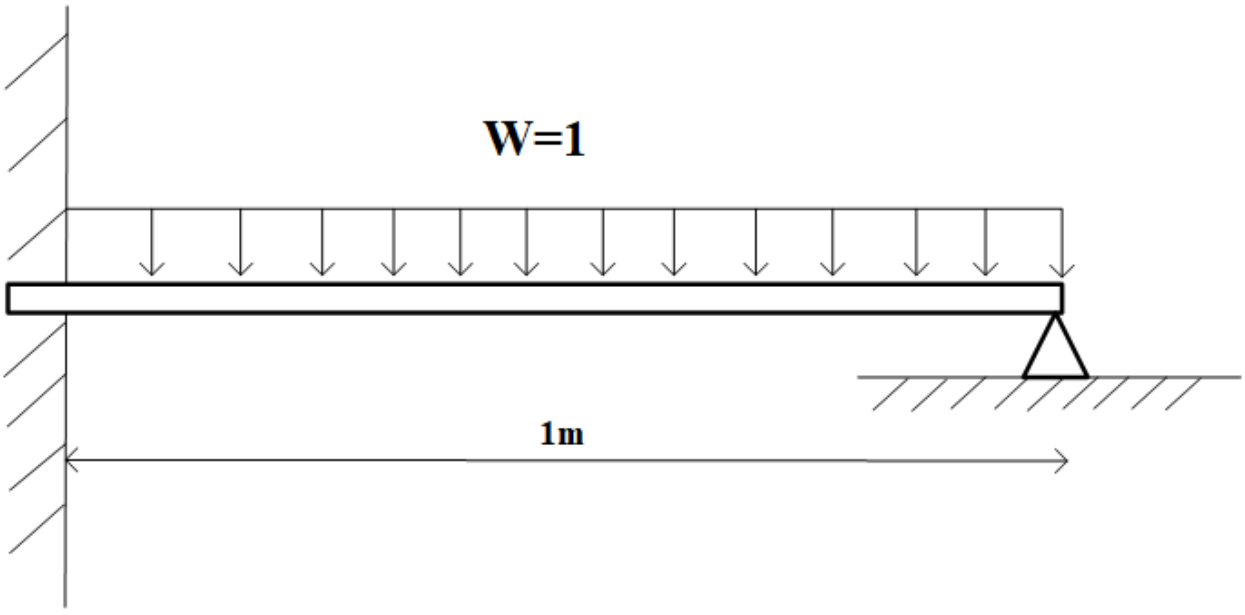
1. When a domain is contained by `Solver.sample_domains`, the `sampling()` will be called every iteration. Users should avoid including redundant domains. Future versions will ignore domains with `constraints={}` in training steps.
2. The current version samples points in memory. When GPU devices are enabled, data exchange between the memory and GPU devices might hinder the performance. In future versions, we will sample points directly in GPU devices if available.

See `examples/simple_poisson`.

2.2 Euler–Bernoulli beam

We consider the Euler–Bernoulli beam equation,

$$ \begin{align} \frac{\partial^{2}}{\partial x^{2}}\left(\frac{\partial^{2} u}{\partial x^{2}}\right)=-1 \ u|*{x=0}=0, u^{\prime}*{x=0}=0, \ u^{\prime \prime}|*{x=1}=0, u^{\prime \prime \prime}*{x=1}=0, \end{align} $$ which models the following beam with external forces.



2.2.1 Expression Node

The Euler-Bernoulli beam equation is not implemented inside IDRLnet. Users may add the equation to `idrlnet.pde_op.equations`. However, one may also define the differential equation via symbol expressions directly.

First, we define a function symbol in the symbol definition part.

```
x = sp.symbols('x')
y = sp.Function('y')(x)
```

In the PDE definition part, we add these PDE nodes:

```
pde1 = sc.ExpressionNode(name='dddd_y', expression=y.diff(x).diff(x).diff(x).diff(x) + 1)
pde2 = sc.ExpressionNode(name='d_y', expression=y.diff(x))
pde3 = sc.ExpressionNode(name='dd_y', expression=y.diff(x).diff(x))
pde4 = sc.ExpressionNode(name='ddd_y', expression=y.diff(x).diff(x).diff(x))
```

These are instances of `idrl.pde.PdeNode`, which are also computational nodes. For example, `pde1` is an instance of `Node` with

- `inputs=tuple()`;
- `derivatives=(y__x__x__x__x, )`;
- `outputs=('dddd_y',)`.

The four PDE nodes match the following operators, respectively:

- dy^4/d^4x+1;
- dy/dx;
- dy^2/d^2x;
- dy^3/d^3x.

2.2.2 Seperate Inference Domain

In this example, we define a domain specified for inference.

```
@sc.datanode(name='infer')
class Infer(sc.SampleDomain):
    def sampling(self, *args, **kwargs):
        return {'x': np.linspace(0, 1, 1000).reshape(-1, 1)}, {}
```

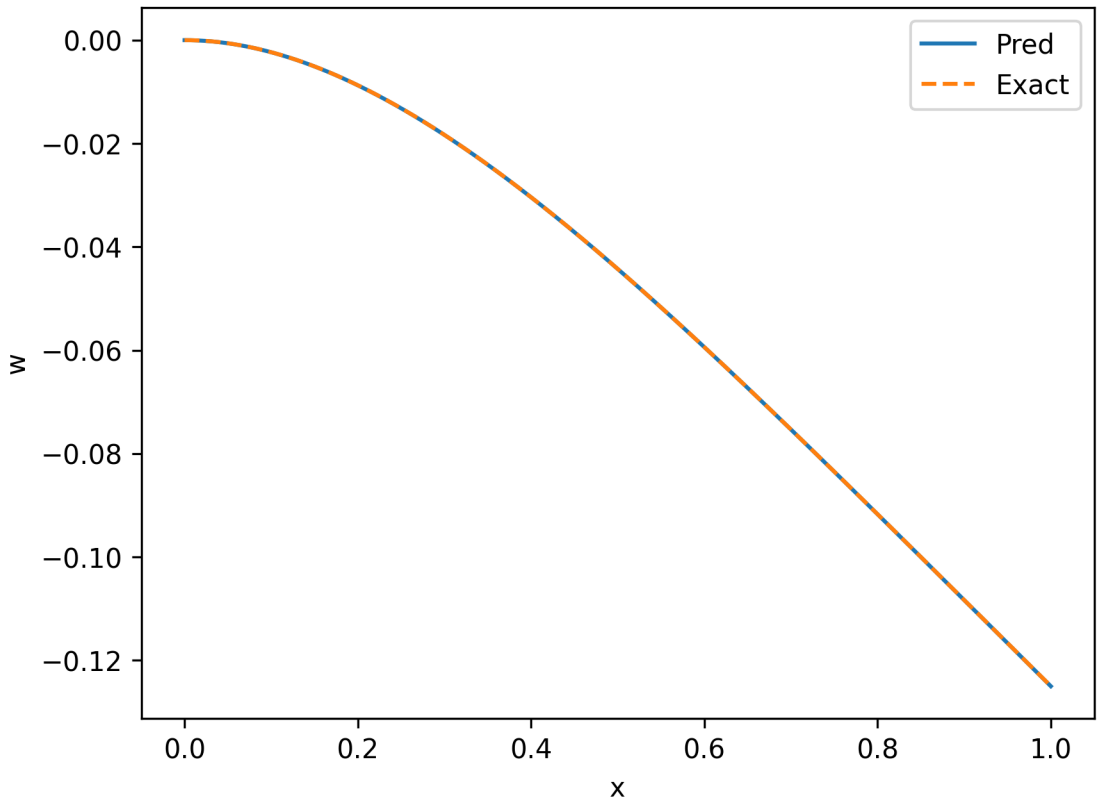
Its instance is not be passed to the solver initializer, which may improve the performance since Infer().sampling After the solving procedure ends, we change the `sample_domains` of the solver,

```
solver.sample_domains = (Infer(),)
```

which triggers the regeneration of the computational graph. Then `solver.infer_step()` is called.

```
points = solver.infer_step({'infer': ['x', 'y']})
xs = points['infer']['x'].detach().cpu().numpy().ravel()
y_pred = points['infer']['y'].detach().cpu().numpy().ravel()
```

The result is shown as follows.



See `examples/euler_beam`.

2.3 Burgers' Equation

Burgers' equation is formulated as following:

$$ \begin{equation} \frac{\partial u}{\partial t}+u \frac{\partial u}{\partial x}=\nu \frac{\partial^{2} u}{\partial x^{2}} \end{equation} $$ We have added the template of the equation into `idrlnet.pde_op.equations`. In this example, we take $\nu=-0.01/\pi$, and the problem is

$$ \begin{equation} \begin{array}{l} u_t+u u_{x}-(0.01 / \pi) u_{x x}=0, \quad x \in[-1,1], \quad t \in[0,1] \ u(0, x)=-\sin (\pi x) \ u(t,-1)=u(t, 1)=0 \end{array} \end{equation}. $$

2.3.1 Time-dependent Domain

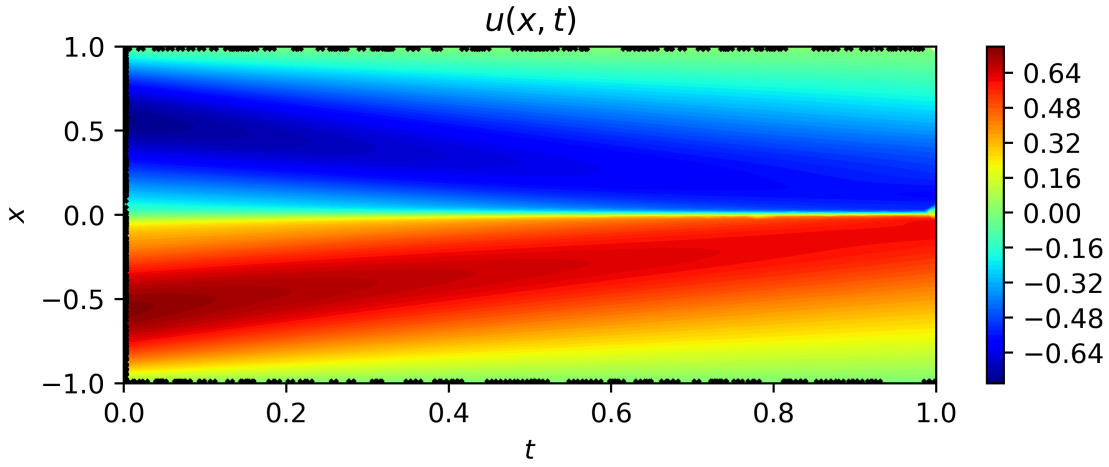
The equation is time-dependent. In addition, we define a time symbol `t` and its range.

```
t_symbol = Symbol('t')
time_range = {t_symbol: (0, 1)}
```

The parameter range `time_range` will be passed to methods `geo.Geometry.sample_interior()` and `geo.Geometry.sample_boundary()`. The sampling methods generate samples containing the additional dims provided in `param_ranges.keys()`.

```
# Interior domain
points = geo.sample_interior(10000, bounds={x: (-1., 1.)}, param_ranges=time_range)

# Initial value condition
points = geo.sample_interior(100, param_ranges={t_symbol: 0.0})

# Boundary condition
points = geo.sample_boundary(100, param_ranges=time_range)
```

The result is shown as follows:

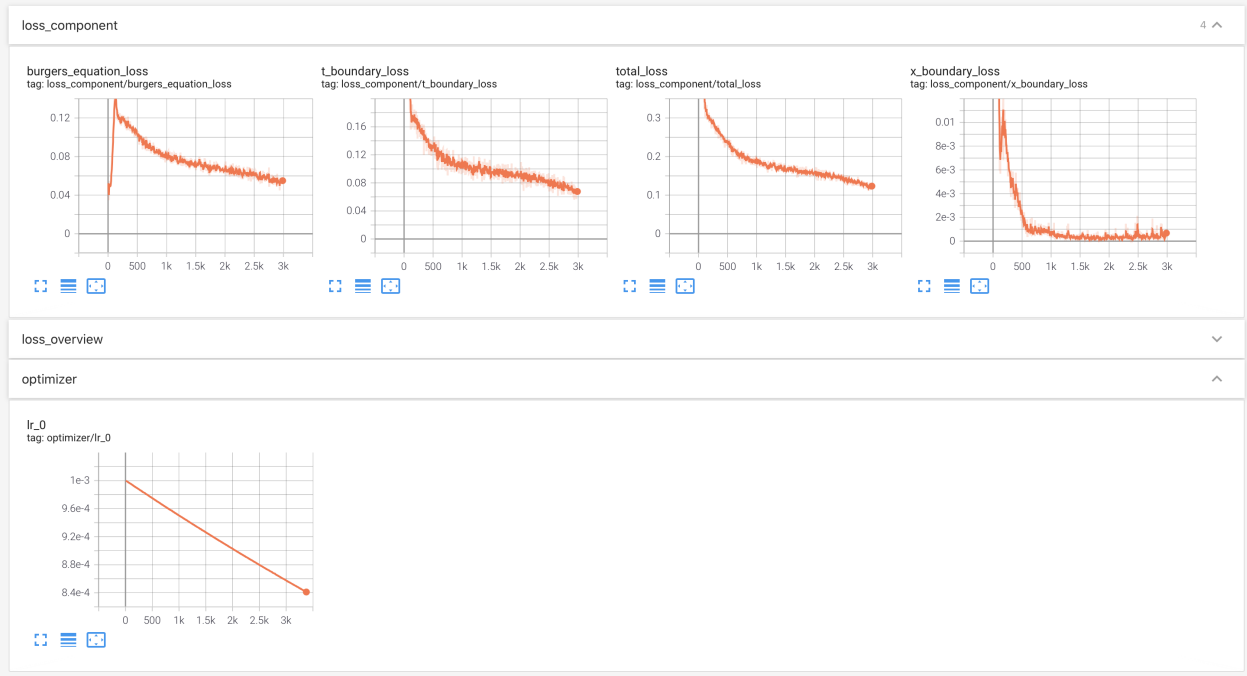


2.3.2 Use TensorBoard

To monitor the training process, we employ TensorBoard. The learning rate, losses on different domains, and the total loss will be recorded automatically. Users can call `Solver.summary_receiver()` to get the instance of `SummaryWriter`. As default, one starts TensorBoard at `./network_idr`:

```
tensorboard --logdir ./network_dir
```

Users can monitor the status of training:



See `examples/burgers_equation`.

2.4 Allen-Cahn Equation

This section repeats the adaptive PINN method presented by Wight and Zhao.

The Allen-Cahn equation has the following general form:

$$ \partial_{t} u=\gamma_{1} \Delta u+\gamma_{2}\left(u-u^{3}\right). $$

Consider the one-dimensional Allen-Cahn equation with periodic boundary conditions:

$$ \begin{array}{l} u_{t}-0.0001 u_{x x}+5 u^{3}-5 u=0, \quad x \in[-1,1], \quad t \in[0,1], \\ u(0, x)=x^{2} \cos (\pi x) \\ u(t,-1)=u(t, 1) \\ u_{x}(t,-1)=u_{x}(t, 1). \end{array} $$

2.4.1 Periodic Boundary Conditions

The periodic boundary conditions are enforced by $u(t, x)=u(t,x+2)$ and $u_x(t, x)=u_x(t,x+2)$ with $x=-1$, which is equivalent to

$$ \begin{array}{l} \tilde u(t,x)=u(t,x+2), \quad \forall t\in[0,1],x\in[-1,1], \\ \tilde u(t,x)=u(t,x),\quad \forall t\in[0,1],x=-1, \\ \tilde u_x(t,x)=u_x(t,x),\quad \forall t\in[0,1],x=-1.\end{array} $$

The transform above is implemented by

```
net_u = sc.MLP([2, 128, 128, 128, 128, 2], activation=sc.Activation.tanh)
net_u = sc.NetNode(inputs=('x', 't',), outputs=('u',), name='net1', net=net_u)
xp = sc.ExpressionNode(name='xp', expression=x + 2)
net_tilde_u = sc.get_shared_net_node(net_u, inputs=('xp', 't',), outputs=('up',), name=
↪'net2', arch='mlp')
```

where `xp` translates x to $x+2$. The node `net_tilde_u` has the same internal parameters as `net_u` while its inputs and outputs are translated.

2.4.2 Receivers acting as Callbacks

We define a group of `Signal` to trigger receivers. They are adequate for customizing various PINN algorithms at the moment.

```
class Signal(Enum):
    REGISTER = 'signal_register'
    SOLVE_START = 'signal_solve_start'
    TRAIN_PIPE_START = 'signal_train_pipe_start'
    AFTER_COMPUTE_LOSS = 'compute_loss'
    BEFORE_BACKWARD = 'signal_before_backward'
    TRAIN_PIPE_END = 'signal_train_pipe_end'
    SOLVE_END = 'signal_solve_end'
```

We implement the adaptive sampling method as follows.

```
class SpaceAdaptiveReceiver(sc.Receiver):
    # implement the abstract method in sc.Receiver
    def receive_notify(self, solver, message):
        # In each iteration, after the train pipe ends, the receiver will be notified.
        # Every five 500 iterations, the adaptive sampling will be triggerd.
        if sc.Signal.TRAIN_PIPE_END in message.keys() and solver.global_step % 1000 == 0:
            sc.logger.info('space adaptive sampling...')
            # Do extra sampling and compute the residual
            results = solver.infer_step({'data_evaluate': ['x', 't', 'sdf', 'AllenCahn_u
↪']})
            residual_data = results['data_evaluate']['AllenCahn_u'].detach().cpu().
↪numpy().ravel()
            # Sort the points by residual loss
            index = np.argsort(-1. * np.abs(residual_data))[:200]
            _points = {key: values[index].detach().cpu().numpy() for key, values in␣
↪results['data_evaluate'].items()}
            _points.pop('AllenCahn_u')
            _points['area'] = np.zeros_like(_points['sdf']) + (1.0 / 200)
            # Update the points in the re_samping_domain
            solver.set_domain_parameter('re_sampling_domain', {'points': _points})
```

We also draw the result every 1000 iterations.

```
class PostProcessReceiver(Receiver):
    def receive_notify(self, solver, message):
        if pinnnet.receivers.Signal.TRAIN_PIPE_END in message.keys() and solver.global_
↪step % 1000 == 1:
            points = s.infer_step({'allen_test': ['x', 't', 'u']})
            triang_total = tri.Triangulation(points['allen_test']['t'].detach().cpu().
↪numpy().ravel(),
                                             points['allen_test']['x'].detach().cpu().
↪numpy().ravel(), )
            plt.tricontourf(triang_total, points['allen_test']['u'].detach().cpu().
↪numpy().ravel(), 100)
            tc_bar = plt.colorbar()
            tc_bar.ax.tick_params(labelsize=12)
            plt.xlabel('$t$')
            plt.ylabel('$x$')
```

(continues on next page)

(continued from previous page)

```
            plt.title('$u(x,t)$')
            plt.savefig(f'result_{solver.global_step}.png')
            plt.show()
```

Before `Solver.solve()` is called, register the two receivers to the solver:

```
s.register_receiver(SpaceAdaptiveReceiver())
s.register_receiver(PostProcessReceiver())
```

The training process is shown as follows:

See `examples/allen_cahn`.

2.5 Inverse Wave Equation

Consider the 1d wave equation:

$$ \begin{equation} \frac{\partial^2u}{\partial t^2}=c^2\frac{\partial^2u}{\partial x^2}, \end{equation} $$ where $c>0$ is unknown and is to be estimated. A group of data pairs ${x_i, t_i, u_i}_{i=1,2,\cdot,N}$ is observed. Then the problem is formulated as:

$$ \min_{u,c} \sum_{i=1,2,\cdots,N} |u(x_i, t_i)-u_i|^2\ s.t. \frac{\partial^2u}{\partial t^2}=c^2\frac{\partial^2u}{\partial x^2} $$

In the context of PINN, u is parameterized to u_θ. The problem above is transformed to the discrete form:

$$ \min_{\theta,c} w_1\sum_{i=1,2,\cdots,N} |u_\theta(x_i, t_i)-u_i|^2 +w_2\sum_{i=1,2,\cdots,M}\left|\frac{\partial^2u_\theta(x_i,t_i)}{\partial t^2}-c^2\frac{\partial^2u_\theta(x_i,t_i)}{\partial x^2}\right|^2. $$

2.5.1 Importing External Data

We take the ground truth

$$ u=\sin x \cdot(\sin 1.54 t + \cos 1.54 t), $$ where $c=1.54$. The external data is generated by

```
    points = geo.sample_interior(density=20,
                                 bounds={x: (0, L)},
                                 param_ranges=time_range,
                                 low_discrepancy=True)
    points['u'] = np.sin(points['x']) * (np.sin(c * points['t']) + np.cos(c * points['t
↪']))

    # Some data points are contaminated.
    points['u'][np.random.choice(len(points['u']), 10, replace=False)] = 3.
```

To use the external data as the data source, we define a data node to store the state:

```
@sc.datanode(name='wave_domain', loss_fn='L1')
class WaveExternal(sc.SampleDomain):
    def __init__(self):
        points = pd.read_csv('external_sample.csv')
```

(continues on next page)

(continued from previous page)

```
        self.points = {col: points[col].to_numpy().reshape(-1, 1) for col in points.
↪columns}
        self.constraints = {'u': self.points['u']}
        self.points.pop('u')

    def sampling(self, *args, **kwargs):
        points = self.points
        constraints = self.constraints
        return points, constraints
```

If large-scale external data are used, users can also implement the `sampling()` method to adapt to external data interfaces.

2.5.2 Define Unknown Parameters

IDRLnet defines a network node with a single parameter to represent the variable.

```
var_c = sc.get_net_node(inputs=('x',), outputs=('c',), arch=sc.Arch.single_var)
```

If bounds for variables are available, users can embed the bounds into the definition.

```
var_c = sc.get_net_node(inputs=('x',), outputs=('c',), arch=sc.Arch.bounded_single_var,␣
↪lower_bound=1., upper_bound=3.0)
```

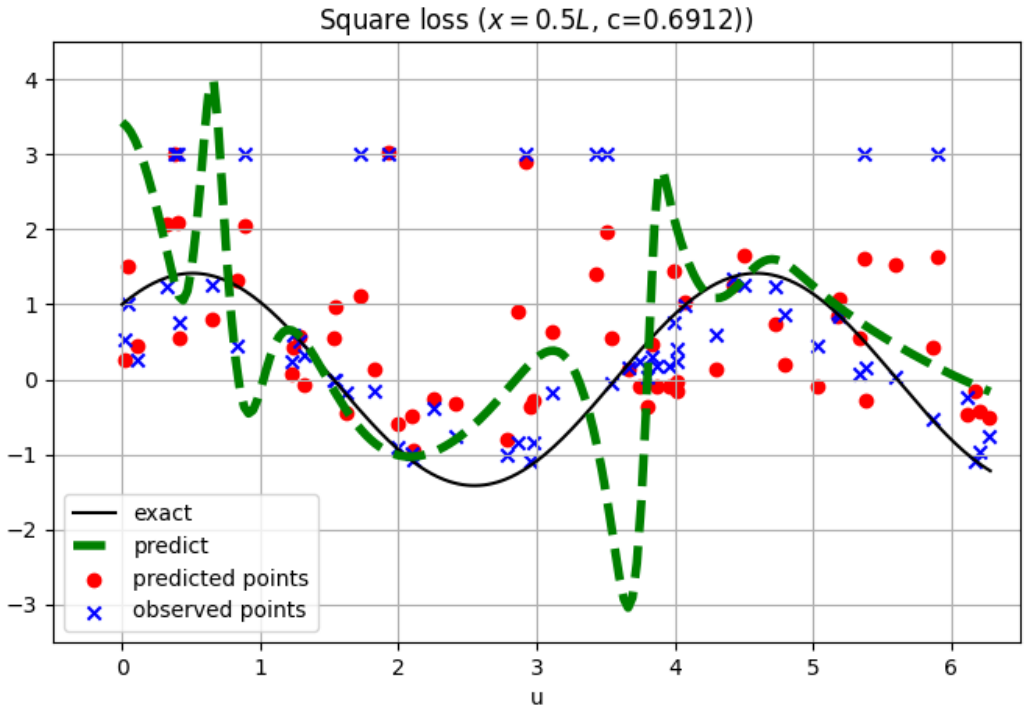
2.5.3 Loss Metrics

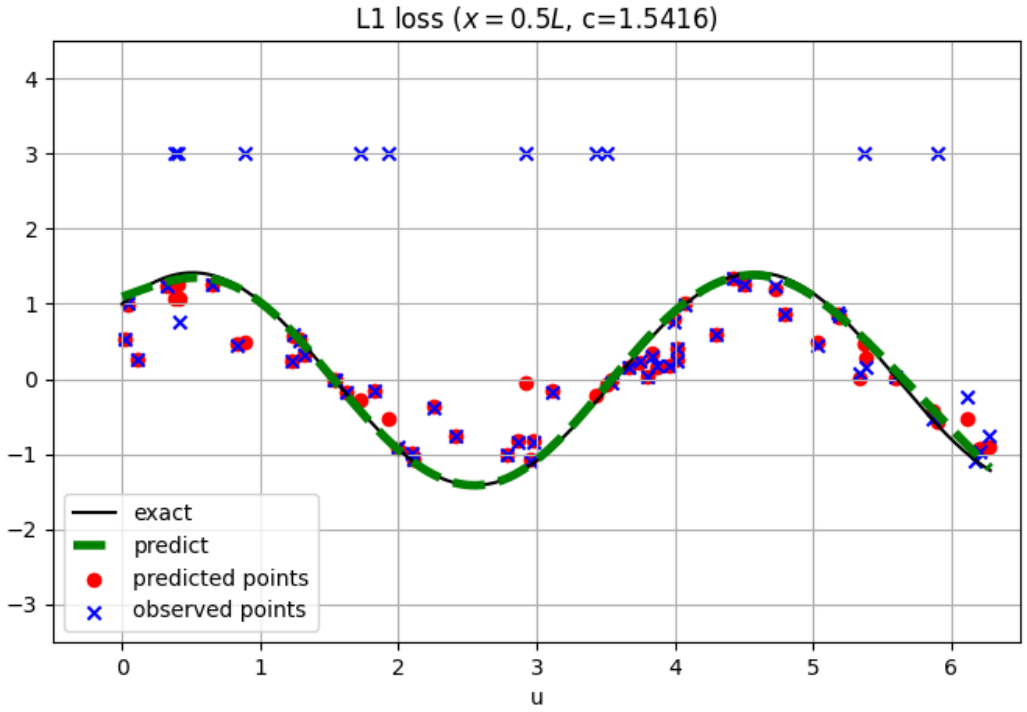
The final loss in each iteration is represented by

$$ loss = \sum_i^M \sigma_i \sum_j^{N_{i}} \lambda_{ij}\times\text{area}_{ij}\times\text{Loss}(y_j, y^{pred}*j), $$ *where M domains are included, and the i-th domain has $N*{i}$* sample points in it.

- By default, The loss function is set to `square`, and the alternative is `L1`. More types will be implemented later.
- area_{ij} is the weight generated by geometric objects automatically.
- σ_i is the weight for the i-th domain loss, which is set to `1.` by default.
- λ_{ij} is the weight for each point.

For robust regression, the `L1` loss is usually preferred over the `square` loss. The conclusion might also hold for inverse PINN as shown:





See `examples/inverse_wave_equation`.

2.6 Parameterized Poisson

We consider an extended problem of *Simple Poisson*.

$$ \begin{array}{l} -\Delta u=1\\ \frac{\partial u(x, -1)}{\partial n}=\frac{\partial u(x, 1)}{\partial n}=0\\ u(-1,y)=T_l\\ u(1, y)=0, \end{array} $$ where T_l is a design parameter ranging in $(-0.2,0.2)$. The target is to train a surrogate that $u_\theta(x,y,T_l)$ gives the temperature at (x,y) when T_l is provided.

2.6.1 Train A Surrogate

In addition, we define the parameter

```
temp = sp.Symbol('temp')
temp_range = {temp: (-0.2, 0.2)}
```

The usage of `temp` is similar to the time variable in *Burgers' Equation*. `temp_range` should be passed to the argument `param_ranges` in sampling domains.

The left bound value condition is

```
@sc.datanode
class Left(sc.SampleDomain):
    # Due to `name` is not specified, Left will be the name of datanode automatically
    def sampling(self, *args, **kwargs):
        points = rec.sample_boundary(1000, sieve=(sp.Eq(x, -1.)), param_ranges=temp_
↪range)
        constraints = sc.Variables({'T': temp})
        return points, constraints
```

The result is shown as follows:

See `examples/parameterized_poisson`.

2.7 Variational Minimization

IDRLnet can solve variational minimization problems. In this section, we try to find a minimal surface of revolution.

Given two points $P_1=(-1, \cosh(-1))$ and $P_2=(0.5, \cosh(0.5))$. Consider a curve $u(x)$ connecting P_1 and P_2. The surface of revolution is generated by rotating the curve with respect to x-axis. This section aims to find the curve that minimizes the surface area. The surface area of revolution is obtained by integrating over cylinders of radius y:

$$ S=\int_{x_1}^{x_2} u(x)\sqrt{u'(x)^2+1}dx. $$

2.7.1 Load a Pretrained Network

IDRLnet supports loading pretrained networks. For faster convergence, we take the initial network to be the segment connecting P_1 and P_2, which is accomplished by fitting the following domain:

```
@sc.datanode(loss_fn='L1')
class Interior(sc.SampleDomain):
    def sampling(self, *args, **kwargs):
        points = geo.sample_interior(100)
        constraints = {'u': (np.cosh(0.5) - np.cosh(-1)) / 1.5 * (x + 1.0) + np.cosh(-1)}
        return points, constraints
```

The training procedure is derivative-free, so it converges quite fast.

Starting another script, we load the network trained above as the initial network.

```
s = sc.Solver(sample_domains=(Boundary(), Interior(), InteriorInfer()),
              netnodes=[net],
              init_network_dirs=['pretrain_network_dir'], # where to find the pretrained␣
↪network
              pdes=[dx_exp, integral, ],
              max_iter=1500)
```

2.7.2 Integral Domain

IDRLnet can calculate definite integration on a domain via Monte Carlo methods.

At the beginning of the script, define `Function` u:

```
u = sp.Function('u')(x)
```

The `ICNode` is responsible for numerical integration. The output of `ICNode` is automatically prefixed with `integral_`. The following code generates a `Node` with output `(integral_dx,)`.

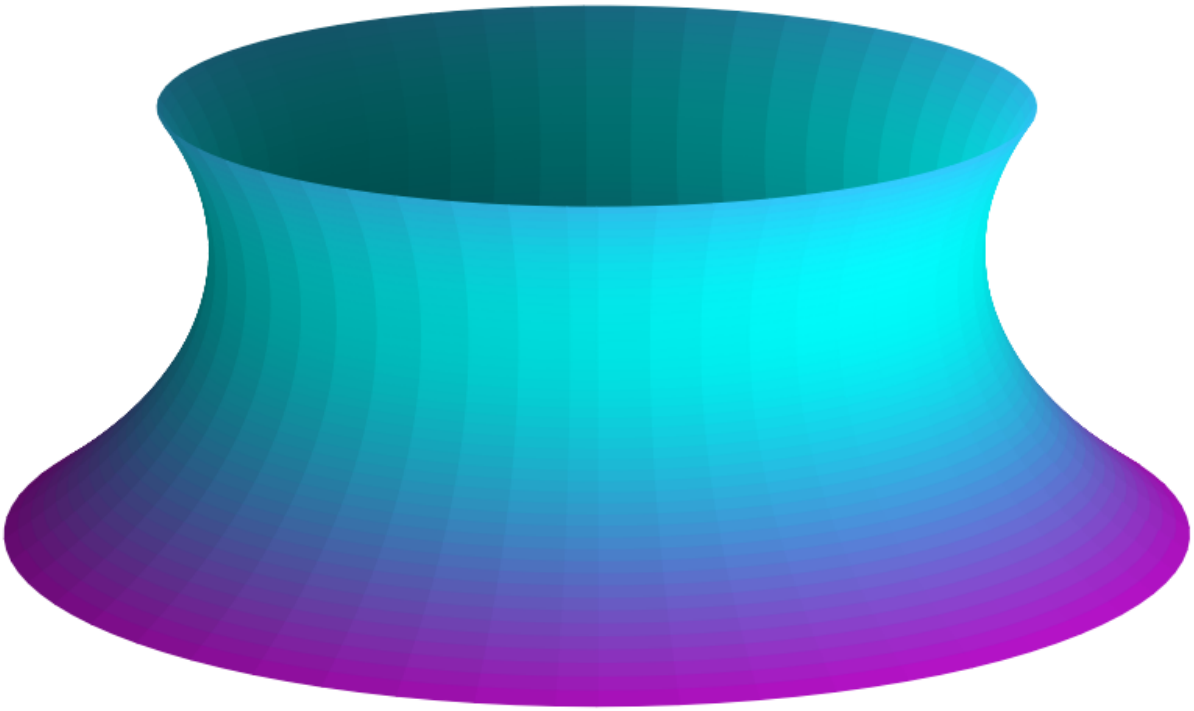
```
dx_exp = sc.ExpressionNode(expression=sp.Abs(u) * sp.sqrt((u.diff(x)) ** 2 + 1), name='dx
↪')
integral = sc.ICNode('dx', dim=1, time=False)
```

Since the minimization model has an obvious lower bound 0, we embed the problem into the constraints:

```
@sc.datanode(loss_fn='L1')
class Interior(sc.SampleDomain):
    def sampling(self, *args, **kwargs):
        points = geo.sample_interior(10000)
        constraints = {'integral_dx': 0, }
        return points, constraints
```

The iterations are show as follows:

The exact solution is:



See `examples/minimal_surface_of_revolution`.

2.8 Volterra Integral Differential Equation

We consider the first-order Volterra type integro-differential equation on $[0, 5]$ (from Lu et al. 2021):

$$ \frac{d y}{d x}+y(x)=\int_{0}^{x} e^{t-x} y(t) d t, \quad y(0)=1 $$ with the ground truth $u=\exp(-x) \cosh x$.

2.8.1 1D integral with Variable Limits

The LHS is represented by

```
exp_lhs = sc.ExpressionNode(expression=f.diff(x) + f, name='lhs')
```

The RHS has an integral with variable limits. Therefore, we introduce the class `Int1DNode`:

```
fs = sp.Symbol('fs')
exp_rhs = sc.Int1DNode(expression=sp.exp(s - x) * fs, var=s, lb=0, ub=x, expression_name=
↪'rhs',
                       funs={'fs': {'eval': netnode,
                                    'input_map': {'x': 's'},
                                    'output_map': {'f': 'fs'}}},
                       degree=10)
```

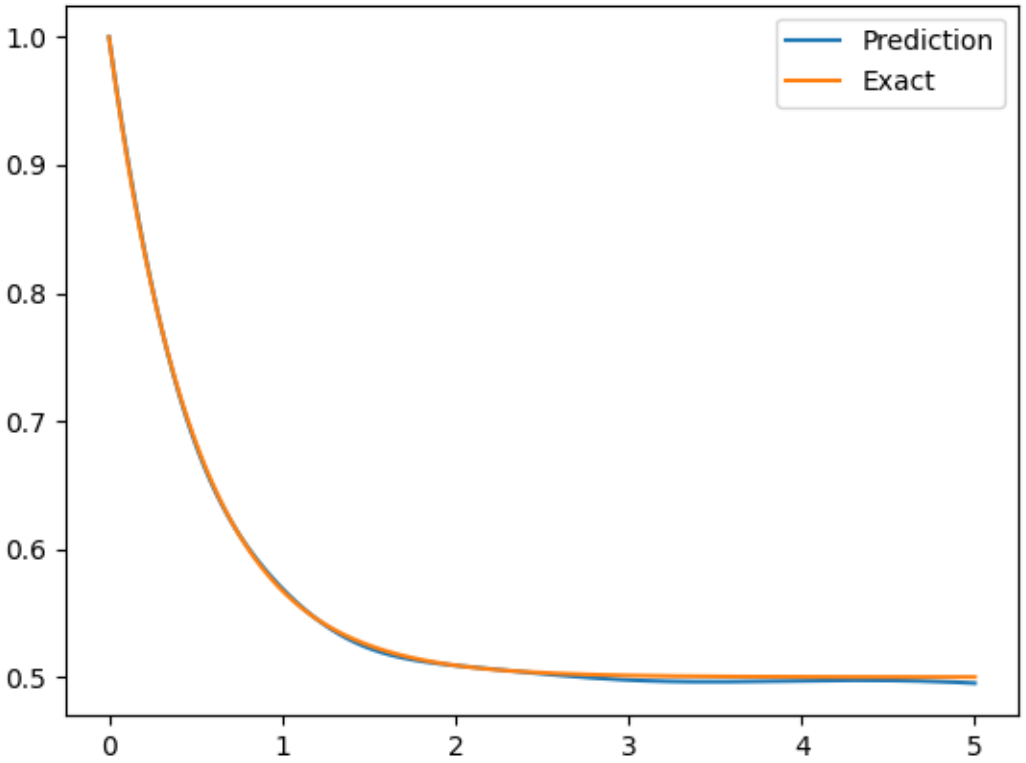
We map `f` and `x` to `fs` and `s` in the integral, respectively. The numerical integration is approximated by Gauss–Legendre quadrature with `degree=10`. The difference between the RHS and the LHS is presented by a `pde_op.opterator.Difference` node,

```
diff = sc.Difference(T='lhs', S='rhs', dim=1, time=False)
```

which generates a node with

- `input=(lhs,rhs)`;
- `output=(difference_lhs_rhs,)`.

The final result is shown as follows:



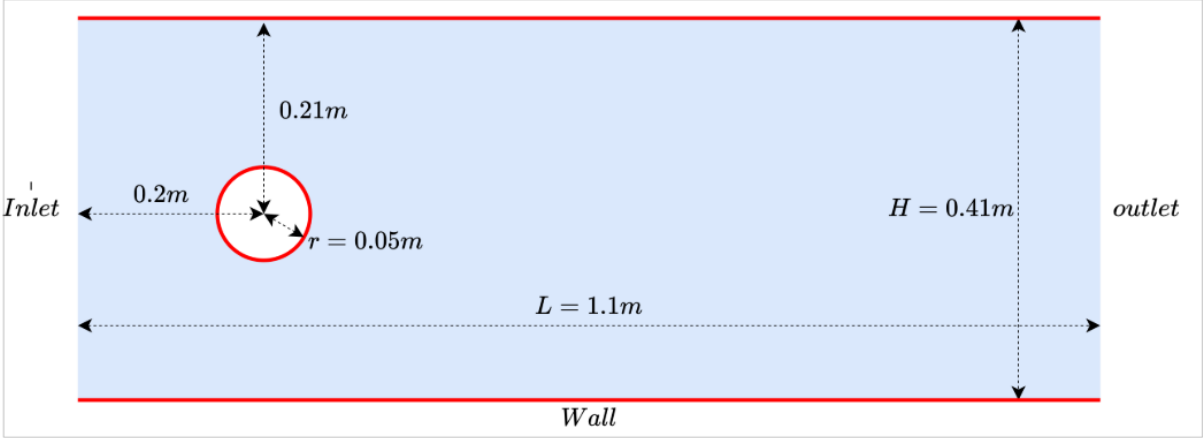
See `examples/Volterra_IDE`.

2.9 Navier-Stokes equations

This section repeats the Robust PINN method presented by Peng et.al.

2.9.1 Steady 2D NS equations

The prototype problem of incompressible flow past a circular cylinder is considered.



The velocity vector is set to zero at all walls and the pressure is set to p = 0 at the outlet. The fluid density is taken as $\rho = 1kg/m^3$ and the dynamic viscosity is taken as $\mu = 2 · 10^{2}kg/m^3$. The velocity profile on the inlet is set as $u(0, y)=4 \frac{U_M}{H^2}(H-y) y$ with $U_M = 1m/s$ and $H = 0.41m$.

The two-dimensional steady-state Navier-Stokes equation is equivalently transformed into the following equations:

$$ \begin{equation} \begin{aligned} \sigma^{11} &=-p+2 \mu u_x \ \sigma^{22} &=-p+2 \mu v_y \ \sigma^{12} &=\mu\left(u_y+v_x\right) \ p &=-\frac{1}{2}\left(\sigma^{11}+\sigma^{22}\right) \ \left(u u_x+v u_y\right) &=\mu\left(\sigma_x^{11}+\sigma_y^{12}\right) \ \left(u v_x+v v_y\right) &=\mu\left(\sigma_x^{12}+\sigma_y^{22}\right) \end{aligned} \end{equation} $$

We construct a neural network with six outputs to satisfy the PDE constraints above:

$$ \begin{equation} u, v, p, \sigma^{11}, \sigma^{12}, \sigma^{22}=net(x, y) \end{equation} $$

Define Symbols and Geometric Objects

For the 2d problem, we define two coordinate symbolsxandy, six variables$ u, v, p, \sigma^{11}, \sigma^{12}, \sigma^{22}$ are defined.

The geometry object is a simple rectangle and circle with the operator -.

```
x = Symbol('x')
y = Symbol('y')
rec = sc.Rectangle((0., 0.), (1.1, 0.41))
cir = sc.Circle((0.2, 0.2), 0.05)
geo = rec - cir
u = sp.Function('u')(x, y)
v = sp.Function('v')(x, y)
p = sp.Function('p')(x, y)
s11 = sp.Function('s11')(x, y)
s22 = sp.Function('s22')(x, y)
s12 = sp.Function('s12')(x, y)
```

Define Sampling Methods and Constraints

For the problem, three boundary conditions , PDE constraint and external data are presented. We use the robust-PINN model inspired by the traditional LAD (Least Absolute Derivation) approach, where the L1 loss replaces the squared L2 data loss.

```
@sc.datanode
class Inlet(sc.SampleDomain):
    def sampling(self, *args, **kwargs):
        points = rec.sample_boundary(1000, sieve=(sp.Eq(x, 0.)))
        constraints = sc.Variables({'u': 4 * (0.41 - y) * y / (0.41 * 0.41)})
        return points, constraints

@sc.datanode
class Outlet(sc.SampleDomain):
    def sampling(self, *args, **kwargs):
        points = geo.sample_boundary(1000, sieve=(sp.Eq(x, 1.1)))
        constraints = sc.Variables({'p': 0.})
        return points, constraints

@sc.datanode
class Wall(sc.SampleDomain):
    def sampling(self, *args, **kwargs):
        points = geo.sample_boundary(1000, sieve=((x > 0.) & (x < 1.1)))
        #print("points3", points)
        constraints = sc.Variables({'u': 0., 'v': 0.})
        return points, constraints

@sc.datanode(name='NS_external')
class Interior_domain(sc.SampleDomain):
    def __init__(self):
        self.density = 2000

    def sampling(self, *args, **kwargs):
        points = geo.sample_interior(2000)
        constraints = {'f_s11': 0., 'f_s22': 0., 'f_s12': 0., 'f_u': 0., 'f_v': 0., 'f_p
↪': 0.}
        return points, constraints

@sc.datanode(name='NS_domain', loss_fn='L1')
class NSExternal(sc.SampleDomain):
    def __init__(self):
        points = pd.read_csv('NSexternel_sample.csv')
        self.points = {col: points[col].to_numpy().reshape(-1, 1) for col in points.
↪columns}
        self.constraints = {'u': self.points.pop('u'), 'v': self.points.pop('v'), 'p':␣
↪self.points.pop('p')}

    def sampling(self, *args, **kwargs):
        return self.points, self.constraints
```

Define Neural Networks and PDEs

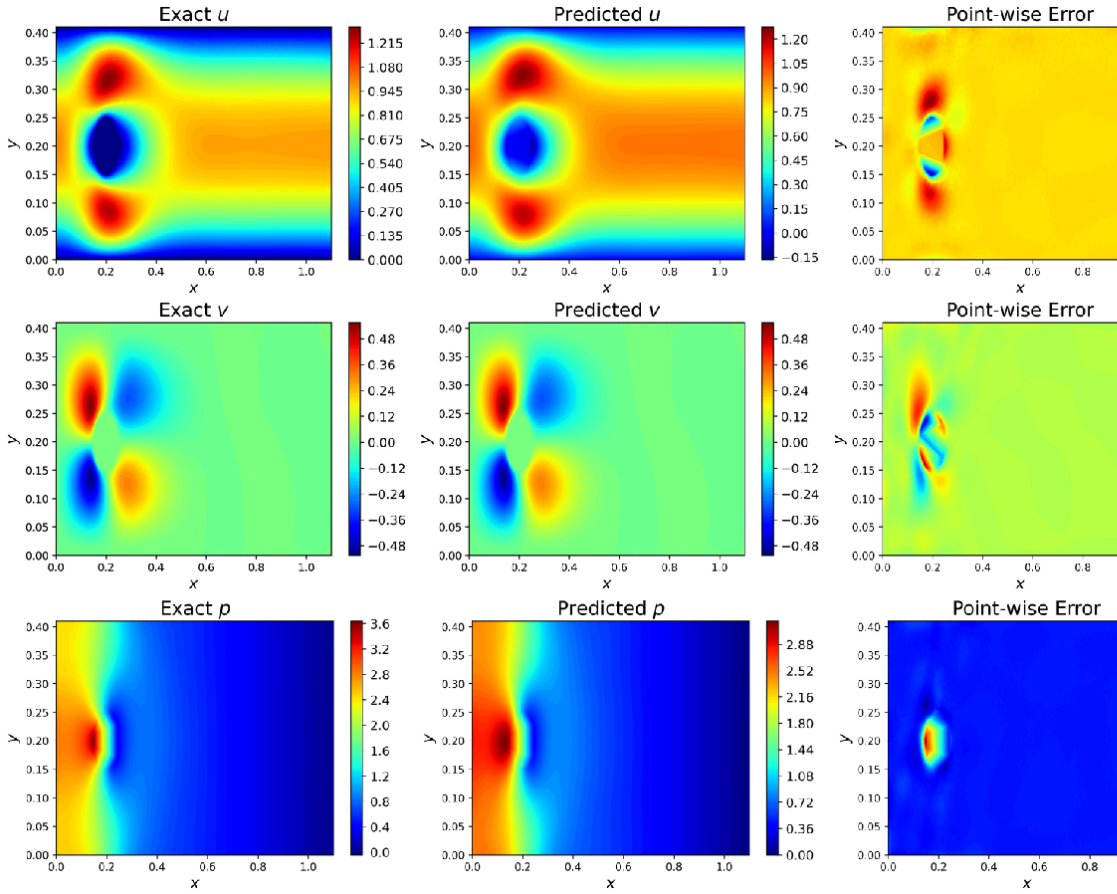
In the PDE definition part, we add these PDE nodes:

```
net = sc.MLP([2, 40, 40, 40, 40, 40, 40, 40, 40, 6], activation=sc.Activation.tanh)
net = sc.get_net_node(inputs=('x', 'y'), outputs=('u', 'v', 'p', 's11', 's22', 's12'),␣
↪name='net', arch=sc.Arch.mlp)
pde1 = sc.ExpressionNode(name='f_s11', expression=-p + 2 * nu * u.diff(x) - s11)
pde2 = sc.ExpressionNode(name='f_s22', expression=-p + 2 * nu * v.diff(y) - s22)
pde3 = sc.ExpressionNode(name='f_s12', expression=nu * (u.diff(y) + v.diff(x)) - s12)
pde4 = sc.ExpressionNode(name='f_u', expression=u * u.diff(x) + v * u.diff(y) - nu *␣
↪(s11.diff(x) + s12.diff(y)))
pde5 = sc.ExpressionNode(name='f_v', expression=u * v.diff(x) + v * v.diff(y) - nu *␣
↪(s12.diff(x) + s22.diff(y)))
pde6 = sc.ExpressionNode(name='f_p', expression=p + (s11 + s22) / 2)
```

Define A Solver

Direct use of Adam optimization is less effective, so the LBFGS optimization method or a combination of both (Adam+LBFGS) is used for training:

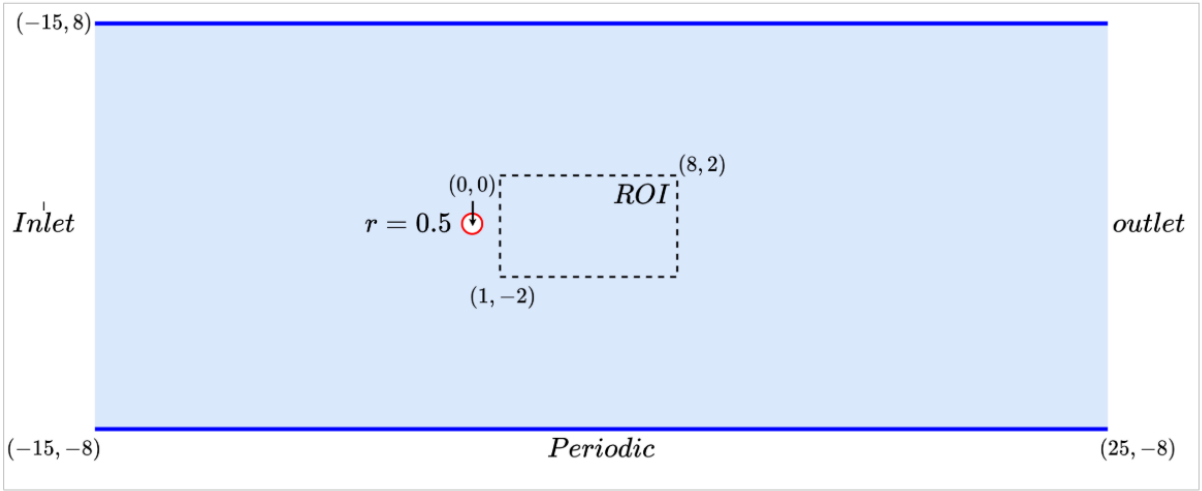
```
s = sc.Solver(sample_domains=(Inlet(), Outlet(), Wall(), Interior_domain(),␣
↪NSExternal()),
              netnodes=[net],
              init_network_dirs=['network_dir_adam'],
              pdes=[pde1, pde2, pde3, pde4, pde5, pde6],
              max_iter=300,
              opt_config = dict(optimizer='LBFGS', lr=1)
             )
```



The result is shown as follows:

2.9.2 Unsteady 2D N-S equations with unknown parameters

A two-dimensional incompressible flow and dynamic vortex shedding past a circular cylinder in a steady-state are numerically simulated. Respectively, the Reynolds number of the incompressible flow is $Re = 100$. The kinematic viscosity of the fluid is $\nu = 0.01$. The cylinder diameter D is 1. The simulation domain size is $[-15,25] × [-8,8]$. The computational domain is much smaller: $[1,8] × [-2,2]× [0,20]$.



$$ \begin{equation} \begin{aligned} &u_t+\lambda_1\left(u u_x+v u_y\right)=-p_x+\lambda_2\left(u_{x x}+u_{y y}\right) \ &v_t+\lambda_1\left(u v_x+v v_y\right)=-p_y+\lambda_2\left(v_{x x}+v_{y y}\right) \end{aligned} \end{equation} $$

where λ_1 and λ_2 are two unknown parameters to be recovered. We make the assumption that $u=\psi_y, \quad v=-\psi_x$

for some stream function $\psi(x, y)$. Under this assumption, the continuity equation will be automatically satisfied. The following architecture is used in this example,

$$ \begin{equation} \psi, p=net\left(t, x, y, \lambda_1, \lambda_2\right) \end{equation} $$

Define Symbols and Geometric Objects

We define three coordinate symbols `x`, `y` and `t`, three variables u,v,p are defined.

```
x = Symbol('x')
y = Symbol('y')
t = Symbol('t')
geo = sc.Rectangle((1., -2.), (8., 2.))
u = sp.Function('u')(x, y, t)
v = sp.Function('v')(x, y, t)
p = sp.Function('p')(x, y, t)
time_range = {t: (0, 20)}
```

Define Sampling Methods and Constraints

This example has only two equation constraints, while the former has six equation constraints. We also use the LAD-PINN model. Then the PDE constrained optimization model is formulated as:

$$ \min *{\theta, \lambda}* \frac{1}{# \mathbf{D}u} \sum*{\left(t_i, x_i, u_i\right) \in \mathbf{D}u}\left|u_i-u\theta\left(t_i, x_i ; \lambda\right)\right|+\omega \cdot L*{p d e} . $$

```
@sc.datanode(name='NS_domain', loss_fn='L1')
class NSExternal(sc.SampleDomain):
    def __init__(self):
        points = pd.read_csv('NSexternel_sample.csv')
        self.points = {col: points[col].to_numpy().reshape(-1, 1) for col in points.
↪columns}
        self.constraints = {'u': self.points.pop('u'), 'v': self.points.pop('v'), 'p':␣
↪self.points.pop('p')}

    def sampling(self, *args, **kwargs):
        return self.points, self.constraints

@sc.datanode(name='NS_external')
class NSEq(sc.SampleDomain):
    def sampling(self, *args, **kwargs):
        points = geo.sample_interior(density=2000, param_ranges=time_range)
        constraints = {'continuity': 0, 'momentum_x': 0, 'momentum_y': 0}
        return points, constraints
```

Define Neural Networks and PDEs

IDRLnet defines a network node to represent the unknown Parameters.

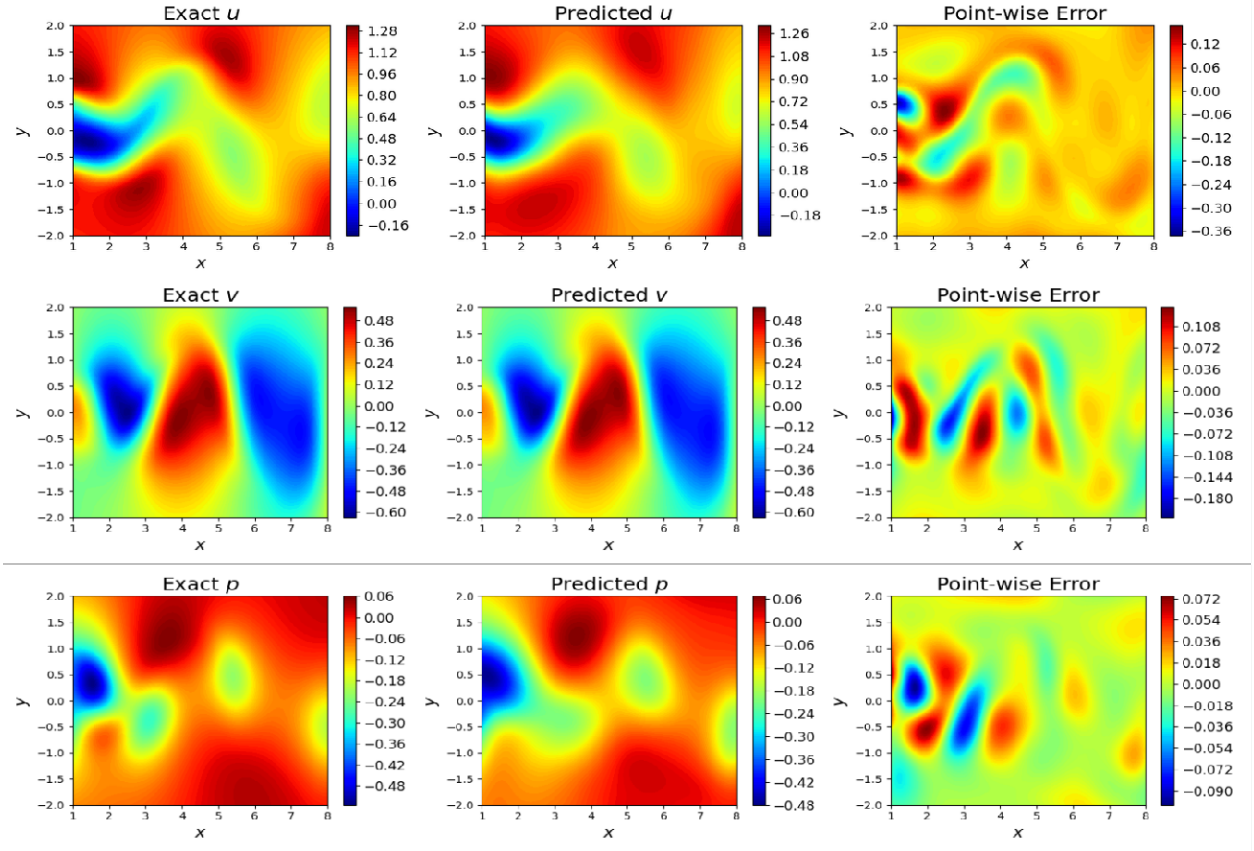
```
net = sc.MLP([3, 20, 20, 20, 20, 20, 20, 20, 20, 3], activation=sc.Activation.tanh)
net = sc.get_net_node(inputs=('x', 'y', 't'), outputs=('u', 'v', 'p'), name='net',␣
↪arch=sc.Arch.mlp)
var_nr = sc.get_net_node(inputs=('x', 'y'), outputs=('nu', 'rho'), arch=sc.Arch.single_
↪var)
pde = sc.NavierStokesNode(nu='nu', rho='rho', dim=2, time=True, u='u', v='v', p='p')
```

Define A Solver

Two nodes trained together

```
s = sc.Solver(sample_domains=(NSExternal(), NSEq()),
              netnodes=[net, var_nr],
              pdes=[pde],
              network_dir='network_dir',
              max_iter=10000)
```

Finally, the real velocity field and pressure field at t=10s are compared with the predicted results:



2.10 Deepritz

This section repeats the Deepritz method presented by Weinan E and Bing Yu.

Consider the 2d Poisson's equation such as the following:

$$ \begin{equation} \begin{aligned} -\Delta u=f, & \text { in } \Omega \ u=0, & \text { on } \partial \Omega \end{aligned} \end{equation} $$

Based on the scattering theorem, its weak form is that both sides are multiplied by$ v \in H_0^1$(which can be interpreted as some function bounded by 0),to get

$$ \int f v=-\int v \Delta u=(\nabla u, \nabla v) $$

The above equation holds for any $v \in H_0^1$. The bilinear part of the right-hand side of the equation with respect to u,v is symmetric and yields the bilinear term:

$$ a(u, v)=\int \nabla u \cdot \nabla v $$

By the Poincaré inequality, the $a(\cdot, \cdot)$ is a positive definite operator. By positive definite, we mean that there exists $\alpha >0$, such that

$$ a(u, u) \geq \alpha|u|^2, \quad \forall u \in H_0^1 $$

The remaining term is a linear generalization of v, which is $l(v)$, which yields the equation:

$$ a(u, v) = l(v) $$

For this equation, by discretizing u,v in the same finite dimensional subspace, we can obtain a symmetric positive definite system of equations, which is the family of Galerkin methods, or we can transform it into a polarization problem to solve it.

To find u satisfies

$$ a(u, v) = l(v), \quad \forall v \in H_0^1 $$

For a symmetric positive definite a , which is equivalent to solving the variational minimization problem, that is, finding u, such that holds, where

$$ J(u) = \frac{1}{2} a(u, u) - l(u) $$

Specifically

$$ \min _{u \in H_0^1} J(u)=\frac{1}{2} \int|\nabla u|_2^2-\int f v $$

The DeepRitz method is similar to the PINN approach, replacing the neural network with u, and after sampling the region, just solve it with a solver like Adam. Written as

$$ \begin{equation} \min *{\left.\hat{u}\right|{\partial \Omega}=0} \hat{J}(\hat{u})=\frac{1}{2} \frac{S_{\Omega}}{N_{\Omega}} \sum\left|\nabla \hat{u}\left(x_i, y_i\right)\right|2^2-\frac{S}{N_{\partial \Omega}} \sum f\left(x_i, y_i\right) \hat{u}\left(x_i, y_i\right) \end{equation} $$

Note that the original $u \in H_0^1$, which is zero on the boundary, is transformed into an unconstrained problem by adding the penalty function term:

$$ \begin{equation} \begin{gathered} \min \hat{J}(\hat{u})=\frac{1}{2} \frac{S_{\Omega}}{N_{\Omega}} \sum\left|\nabla \hat{u}\left(x_i, y_i\right)\right|2^2-\frac{S}{N_{\Omega}} \sum f\left(x_i, y_i\right) \hat{u}\left(x_i, y_i\right)+\beta \frac{S_{\partial \Omega}}{N_{\partial \Omega}} \ \sum \hat{u}^2\left(x_i, y_i\right) \end{gathered} \end{equation} $$

Consider the 2d Poisson's equation defined on $\Omega=[-1,1]\times[-1,1]$, which satisfies $f=2 \pi^2 \sin (\pi x) \sin (\pi y)$.

2.10.1 Define Sampling Methods and Constraints

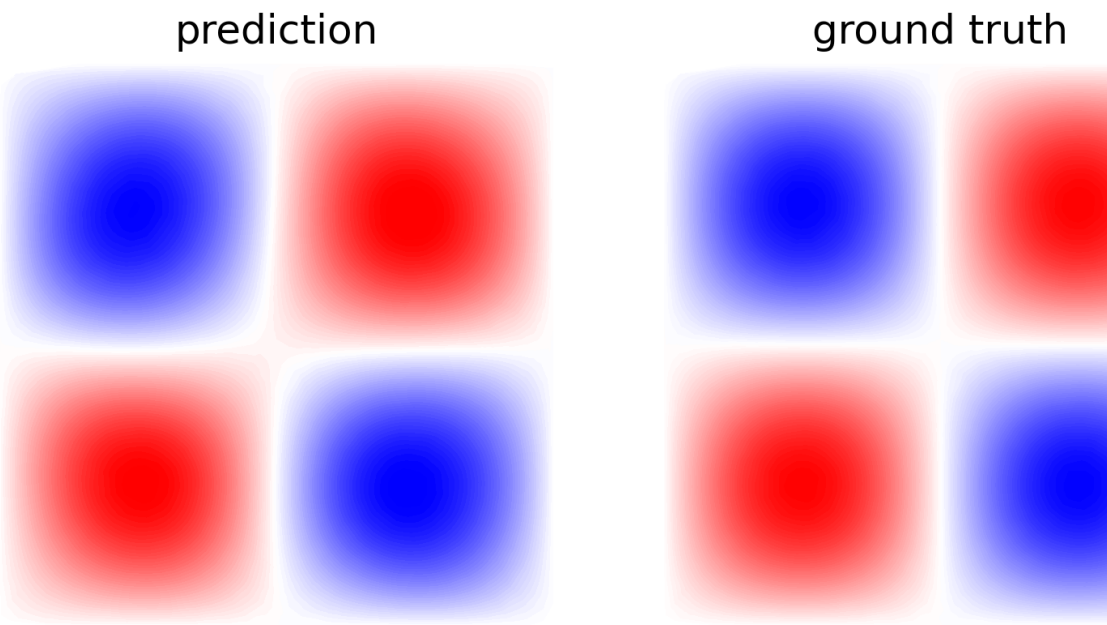
For the problem, boundary condition and PDE constraint are presented and use the Identity loss.

```
@sc.datanode(sigma=1000.0)
class Boundary(sc.SampleDomain):
    def __init__(self):
        self.points = geo.sample_boundary(100,)
        self.constraints = {"u": 0.}

    def sampling(self, *args, **kwargs):
        return self.points, self.constraints


@sc.datanode(loss_fn="Identity")
class Interior(sc.SampleDomain):
    def __init__(self):
        self.points = geo.sample_interior(1000)
        self.constraints = {"integral_dxdy": 0,}

    def sampling(self, *args, **kwargs):
        return self.points, self.constraints
```

2.10.2 Define Neural Networks and PDEs

In the PDE definition section, based on the DeepRitz method we add two types of PDE nodes:

```
def f(x, y):
    return 2 * sp.pi ** 2 * sp.sin(sp.pi * x) * sp.sin(sp.pi * y)

dx_exp = sc.ExpressionNode(
    expression=0.5*(u.diff(x) ** 2 + u.diff(y) ** 2) - u * f(x, y), name="dxdy"
)
net = sc.get_net_node(inputs=("x", "y"), outputs=("u",), name="net", arch=sc.Arch.mlp)

integral = sc.ICNode("dxdy", dim=2, time=False)
```



The result is shown as follows:

CHAPTER

THREE

CITE IDRLNET

```
@misc{peng2021idrlnet,
      title={IDRLnet: A Physics-Informed Neural Network Library},
      author={Wei Peng and Jun Zhang and Weien Zhou and Xiaoyu Zhao and Wen Yao and␣
↪Xiaoqian Chen},
      year={2021},
      eprint={2107.04320},
      archivePrefix={arXiv},
      primaryClass={cs.LG}
}
```

CHAPTER

FOUR

THE TEAM

IDRLnet was developed by members of IDRL laboratory.

CHAPTER

FIVE

FEATURES

IDRLnet is a machine learning library on top of PyTorch. Use IDRLnet if you need a machine learning library that solves both forward and inverse differential equations via physics-informed neural networks (PINN). IDRLnet is a flexible framework inspired by Nvidia Simnet.

IDRLnet supports

- complex domain geometries without mesh generation. Provided geometries include interval, triangle, rectangle, polygon, circle, sphere... Other geometries can be constructed using three boolean operations: union, difference, and intersection;
- sampling in the interior of the defined geometry or on the boundary with given conditions.
- enables the user code to be structured. Data sources, operations, constraints are all represented by `Node`. The graph will be automatically constructed via label symbols of each node. Getting rid of the explicit construction via explicit expressions, users model problems more naturally.
- solving variational minimization problem;
- solving integral differential equation;
- adaptive resampling;
- recover unknown parameter of PDEs from noisy measurement data.

CHAPTER

SIX

API REFERENCE

If you are looking for usage of a specific function, class or method, please refer to the following part.

6.1 idrlnet

6.1.1 idrlnet package

idrlnet.**use_cpu**()

Use CPU.

idrlnet.**use_gpu**(*device=0*)

Use GPU with device *device*.

Parameters

device (*torch.device or int*) – selected device.

Subpackages

idrlnet.architecture package

Submodules

idrlnet.architecture.grid module

The module is experimental. It may be removed or totally refactored in the future.

class idrlnet.architecture.grid.**Interface**(*points1*, *points2*, *nr*, *outputs*, *i1*, *j1*, *i2*, *j2*, *overlap=0.2*)

Bases: object

class idrlnet.architecture.grid.**NetEval**(*n_inputs: int*, *n_outputs: int*, *columns*, *rows*, ***kwargs*)

Bases: Module

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 idrlnet.architecture.grid.**NetGridNode**(*inputs: Union[Tuple, List[str]]*, *outputs: Union[Tuple, List[str]]*, *x_segments: Optional[List[float]] = None*, *y_segments: Optional[List[float]] = None*, *z_segments: Optional[List[float]] = None*, *t_segments: Optional[List[float]] = None*, *columns: Optional[List[float]] = None*, *rows: Optional[List[float]] = None*, **args*, ***kwargs*)

Bases: *NetNode*

get_grid(*overlap*, *nr_points_per_interface_area=100*)

idrlnet.architecture.grid.**get_net_reg_grid**(*inputs: Union[Tuple, List[str]]*, *outputs: Union[Tuple, List[str]]*, *name: str*, *x_segments: Optional[List[float]] = None*, *y_segments: Optional[List[float]] = None*, *z_segments: Optional[List[float]] = None*, *t_segments: Optional[List[float]] = None*, ***kwargs*)

idrlnet.architecture.grid.**get_net_reg_grid_2d**(*inputs: Union[Tuple, List[str]]*, *outputs: Union[Tuple, List[str]]*, *name: str*, *columns: List[float]*, *rows: List[float]*, ***kwargs*)

idrlnet.architecture.grid.**indicator**(*xn: Tensor*, **axis_bounds*)

idrlnet.architecture.layer module

The module provide elements for construct MLP.

class idrlnet.architecture.layer.**Activation**(*value*)

Bases: Enum

An enumeration.

leaky_relu = 'leaky_relu'

poly = 'poly'

relu = 'relu'

selu = 'selu'

sigmoid = 'sigmoid'

silu = 'silu'

sin = 'sin'

swish = 'swish'

tanh = 'tanh'

class idrlnet.architecture.layer.**Initializer**(*value*)

Bases: Enum

An enumeration.

Xavier_uniform = 'Xavier_uniform'

constant = 'constant'

default = 'default'

kaiming_uniform = 'kaiming_uniform'

idrlnet.architecture.layer.**get_activation_layer**(*activation: Activation = Activation.swish*, **args*, ***kwargs*)

idrlnet.architecture.layer.**get_linear_layer**(*input_dim: int*, *output_dim: int*, *weight_norm=False*, *initializer: Initializer = Initializer.Xavier_uniform*, **args*, ***kwargs*)

idrlnet.architecture.mlp module

This module provide some MLP architectures.

class idrlnet.architecture.mlp.**Arch**(*value*)

Bases: Enum

Enumerate pre-defined neural networks.

bounded_single_var = 'bounded_single_var'

mlp = 'mlp'

mlp_xl = 'mlp_xl'

single_var = 'single_var'

siren = 'siren'

toy = 'toy'

class idrlnet.architecture.mlp.**BoundedSingleVar**(*lower_bound*, *upper_bound*)

Bases: Module

Wrapper a single parameter to represent an unknown coefficient in inverse problem with the upper and lower bound.

Parameters

- **lower_bound** (*float*) – The lower bound for the parameter.
- **upper_bound** (*float*) – The upper bound for the parameter.

forward(*x*) → Tensor

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.

get_value() → Tensor

training: bool

class idrlnet.architecture.mlp.**MLP**(*n_seq: List[int]*, *activation: Union[*Activation*, List[*Activation*]] = Activation.swish*, *initialization:* Initializer *= Initializer.kaiming_uniform*, *weight_norm: bool = True*, *name: str = 'mlp'*, **args*, ***kwargs*)

Bases: `Module`

A subclass of torch.nn.Module customizes a multiple linear perceptron network.

Parameters

- **n_seq** (*List[int]*) – Define neuron numbers in each layer. The number of the first and the last should be in keeping with inputs and outputs.
- **activation** (*Union[*Activation*,List[*Activation*]]*) – By default, the activation is *Activation.swish*.
- **initialization** –

:type initialization:Initializer :param weight_norm: If weight normalization is used. :type weight_norm: bool :param name: Symbols will appear in the name of each layer. Do not confuse with the netnode name. :type name: str :param args: :param kwargs:

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 idrlnet.architecture.mlp.**SimpleExpr**(*expr*, *name='expr'*)

Bases: `Module`

This class is for testing. One can override SimpleExper.forward to represent complex formulas.

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 idrlnet.architecture.mlp.**SingleVar**(*initialization: float = 1.0*)

Bases: Module

Wrapper a single parameter to represent an unknown coefficient in inverse problem.

Parameters

initialization (*float*) – initialization value for the parameter. The default is 0.01

forward(*x*) → Tensor

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.

---

get_value() → Tensor

training: bool

class idrlnet.architecture.mlp.**Siren**(*n_seq: List[int]*, *first_omega: float = 30.0*, *omega: float = 30.0*, *name: str = 'siren'*, **args*, ***kwargs*)

Bases: Module

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.

---

static get_siren_layer(*input_dim: int*, *output_dim: int*, *is_first: bool*, *omega_0: float*)

training: bool

idrlnet.architecture.mlp.**get_inter_name**(*length: int*, *prefix: str*)

idrlnet.architecture.mlp.**get_net_node**(*inputs: Union[Tuple[str, ...], List[str]]*, *outputs: Union[Tuple[str, ...], List[str]]*, *arch: Optional[Arch] = None*, *name=None*, **args*, ***kwargs*) → *NetNode*

Get a net node wrapping networks with pre-defined configurations

Parameters

- **inputs** (*Union[Tuple[str, ...]*) – Input symbols for the generated node.
- **outputs** (*Union[Tuple[str, ...]*) – Output symbols for the generated node.
- **arch** (Arch) – One can choose one of - Arch.mlp - Arch.mlp_xl(more layers and more neurons) - Arch.single_var - Arch.bounded_single_var
- **name** (*str*) – The name of the generated node.

- **args** –
- **kwargs** –

Returns

idrlnet.architecture.mlp.**get_shared_net_node**(*shared_node: NetNode*, *inputs: Union[Tuple[str, ...], List[str]]*, *outputs: Union[Tuple[str, ...], List[str]]*, *name=None*, **args*, ***kwargs*) → *NetNode*

Construct a netnode, the net of which is shared by a given netnode. One can specify different inputs and outputs just like an independent netnode. However, the net parameters may have multiple references. Thus the step operations during optimization should only be applied once.

Parameters

- **shared_node** (*NetNode*) – An existing netnode, the network of which will be shared.
- **inputs** (*Union[Tuple[str, ...]]*) – Input symbols for the generated node.
- **outputs** (*Union[Tuple[str, ...]]*) – Output symbols for the generated node.
- **name** (*str*) – The name of the generated node.
- **args** –
- **kwargs** –

Returns

idrlnet.geo_utils package

Submodules

idrlnet.geo_utils.geo module

This module defines basic behaviour of Geometric Objects.

class idrlnet.geo_utils.geo.**AbsCheckMix**(*name*, *bases*, *namespace*, ***kwargs*)

Bases: ABCMeta, *CheckMeta*

class idrlnet.geo_utils.geo.**AbsGeoObj**

Bases: object

abstract rotation(*angle: float*, *axis: str = 'z'*)

abstract scaling(*scale: float*)

abstract translation(*direction*)

class idrlnet.geo_utils.geo.**CheckMeta**

Bases: type

Make sure that elements are checked when an instance is created,

class idrlnet.geo_utils.geo.**Edge**(*functions*, *ranges: Dict*, *area*)

Bases: *AbsGeoObj*

property axes: List[str]

rotation(*angle: float*, *axis: str = 'z'*)

sample(*density: int*, *param_ranges=None*, *low_discrepancy=False*) → Dict[str, ndarray]

scaling(*scale: float*)

translation(*direction*)

class idrlnet.geo_utils.geo.**Geometry**(**args*, ***kwargs*)

Bases: *AbsGeoObj*

property axes: List[str]

bounds: Dict = None

check_elements()

duplicate() → *Geometry*

edges: List[*Edge*] = None

generate_geo_obj(*other=None*)

rotation(*angle: float*, *axis: str = 'z'*, *center=None*) → *Geometry*

sample_boundary(*density: int*, *sieve=None*, *param_ranges: Optional[Dict] = None*, *low_discrepancy=False*) → Dict[str, ndarray]

sample_interior(*density: int*, *bounds: Optional[Dict] = None*, *sieve=None*, *param_ranges: Optional[Dict] = None*, *low_discrepancy=False*) → Dict[str, ndarray]

scaling(*scale: float*, *center: Optional[Tuple] = None*) → *Geometry*

sdf = None

translation(*direction: Union[List, Tuple]*) → *Geometry*

class idrlnet.geo_utils.geo.**Geometry1D**(**args*, ***kwargs*)

Bases: *Geometry*

class idrlnet.geo_utils.geo.**Geometry2D**(**args*, ***kwargs*)

Bases: *Geometry*

class idrlnet.geo_utils.geo.**Geometry3D**(**args*, ***kwargs*)

Bases: *Geometry*

idrlnet.geo_utils.geo_builder module

A simple factory for constructing Geometric Objects

class idrlnet.geo_utils.geo_builder.**GeometryBuilder**

Bases: object

```
GEOMAP = {'Box': <class 'idrlnet.geo_utils.geo_obj.Box'>, 'Channel': <class
'idrlnet.geo_utils.geo_obj.Tube3D'>, 'Channel2D': <class
'idrlnet.geo_utils.geo_obj.Tube2D'>, 'Channel3D': <class
'idrlnet.geo_utils.geo_obj.Tube3D'>, 'Circle': <class
'idrlnet.geo_utils.geo_obj.Circle'>, 'CircularTube': <class
'idrlnet.geo_utils.geo_obj.CircularTube'>, 'Cylinder': <class
'idrlnet.geo_utils.geo_obj.Cylinder'>, 'Heart': <class
'idrlnet.geo_utils.geo_obj.Heart'>, 'Line': <class
'idrlnet.geo_utils.geo_obj.Line'>, 'Line1D': <class
'idrlnet.geo_utils.geo_obj.Line1D'>, 'Plane': <class
'idrlnet.geo_utils.geo_obj.Plane'>, 'Rectangle': <class
'idrlnet.geo_utils.geo_obj.Rectangle'>, 'Sphere': <class
'idrlnet.geo_utils.geo_obj.Sphere'>, 'Triangle': <class
'idrlnet.geo_utils.geo_obj.Triangle'>}
```

static get_geometry(*geo: str*, ***kwargs*) → *Geometry*

Simple factory method for constructing geometry object. :param geo: Specified a string for geometry, which should be in GeometryBuilder.GEOMAP :rtype geo: str :param kwargs: :return: A geometry object with given kwargs. :rtype: Geometry

idrlnet.geo_utils.geo_obj module

Concrete shape.

class idrlnet.geo_utils.geo_obj.**Box**(**args*, ***kwargs*)

Bases: *Geometry3D*

class idrlnet.geo_utils.geo_obj.**Circle**(**args*, ***kwargs*)

Bases: *Geometry2D*

class idrlnet.geo_utils.geo_obj.**CircularTube**(**args*, ***kwargs*)

Bases: *Geometry3D*

class idrlnet.geo_utils.geo_obj.**Cylinder**(**args*, ***kwargs*)

Bases: *Geometry3D*

class idrlnet.geo_utils.geo_obj.**Heart**(**args*, ***kwargs*)

Bases: *Geometry2D*

class idrlnet.geo_utils.geo_obj.**Line**(**args*, ***kwargs*)

Bases: *Geometry2D*

class idrlnet.geo_utils.geo_obj.**Line1D**(**args*, ***kwargs*)

Bases: *Geometry1D*

class idrlnet.geo_utils.geo_obj.**Plane**(**args*, ***kwargs*)

Bases: *Geometry3D*

class idrlnet.geo_utils.geo_obj.**Polygon**(**args*, ***kwargs*)

Bases: *Geometry2D*

rotation(*angle: float*, *axis: str = 'z'*, *center=None*)

scaling(*scale: float*, *center: Optional[Tuple] = None*)

translation(*direction: Union[List, Tuple]*)

class idrlnet.geo_utils.geo_obj.**Rectangle**(**args*, ***kwargs*)

Bases: *Geometry2D*

class idrlnet.geo_utils.geo_obj.**Sphere**(**args*, ***kwargs*)

Bases: *Geometry3D*

class idrlnet.geo_utils.geo_obj.**Triangle**(**args*, ***kwargs*)

Bases: *Geometry2D*

class idrlnet.geo_utils.geo_obj.**Tube**(**args*, ***kwargs*)

Bases: *Tube3D*

class idrlnet.geo_utils.geo_obj.**Tube2D**(**args*, ***kwargs*)

Bases: *Geometry2D*

class idrlnet.geo_utils.geo_obj.**Tube3D**(**args*, ***kwargs*)

Bases: *Geometry3D*

idrlnet.geo_utils.sympy_np module

Convert sympy expression to np functions todo: converges to torch_util

idrlnet.geo_utils.sympy_np.**lambdify_np**(*f*, *r: Iterable*)

idrlnet.pde_op package

Submodules

idrlnet.pde_op.equations module

Predefined equations

class idrlnet.pde_op.equations.**AllenCahnNode**(*u='u'*, *gamma_1=0.0001*, *gamma_2=5*)

Bases: *PdeNode*

class idrlnet.pde_op.equations.**BurgersNode**(*u: str = 'u'*, *v='v'*)

Bases: *PdeNode*

class idrlnet.pde_op.equations.**DiffusionNode**(*T='T'*, *D='D'*, *Q=0*, *dim=3*, *time=True*, ***kwargs*)

Bases: *PdeNode*

class idrlnet.pde_op.equations.**NavierStokesNode**(*nu=0.1*, *rho=1.0*, *dim=2.0*, *time=False*, ***kwargs*)

Bases: *PdeNode*

class idrlnet.pde_op.equations.**SchrodingerNode**(*u='u'*, *v='v'*, *c=0.5*)

Bases: *PdeNode*

class idrlnet.pde_op.equations.**WaveNode**(*u='u'*, *c='c'*, *dim=3*, *time=True*, ***kwargs*)

Bases: *PdeNode*

idrlnet.pde_op.operator module

Operators in PDE

class idrlnet.pde_op.operator.**Curl**(*vector*, *curl_name=None*)

Bases: *PdeNode*

class idrlnet.pde_op.operator.**Derivative**(*T: Union[str, Symbol, float, int]*, *p: Union[str, Symbol]*, *S: Union[str, Symbol, float, int] = 0.0*, *dim=3*, *time=True*)

Bases: *PdeNode*

class idrlnet.pde_op.operator.**Difference**(*T: Union[str, Symbol, float, int]*, *S: Union[str, Symbol, float, int]*, *dim=3*, *time=True*)

Bases: *PdeNode*

class idrlnet.pde_op.operator.**Divergence**(*vector*, *div_name='div_v'*)

Bases: *PdeNode*

class idrlnet.pde_op.operator.**ICNode**(*T: Union[str, Symbol, int, float, List[Union[str, Symbol, int, float]]]*, *dim: int = 2*, *time: bool = False*, *reduce_name: Optional[str] = None*)

Bases: *PdeNode*

class idrlnet.pde_op.operator.**Int1DNode**(*expression*, *expression_name*, *lb*, *ub*, *var: Union[str, Symbol] = 's'*, *degree=20*, ***kwargs*)

Bases: *PdeNode*

counter = 0

make_nodes() → None

new_node(*name: Optional[str] = None*, *tf_eq: Optional[Expr] = None*, *free_symbols: Optional[List[str]] = None*, **args*, ***kwargs*)

class idrlnet.pde_op.operator.**IntEq**(*binding_node*, *lb_lambda*, *ub_lambda*, *out_symbols*, *free_symbols*, *eq_lambda*, *name*)

Bases: `object`

class idrlnet.pde_op.operator.**NormalGradient**(*T: Union[str, Symbol, float, int]*, *dim=3*, *time=True*)

Bases: *PdeNode*

Submodules

idrlnet.callbacks module

Basic Callback classes

class idrlnet.callbacks.**GradientReceiver**

Bases: *Receiver*

Register the receiver to monitor gradient norm on the Tensorboard.

receive_notify(*solver:* Solver, *message*)

class idrlnet.callbacks.**HandleResultReceiver**(*result_dir*)

Bases: *Receiver*

The receiver will be automatically registered to save results on training domains.

receive_notify(*solver:* Solver, *message: Dict*)

class idrlnet.callbacks.**SummaryReceiver**(**args*, ***kwargs*)

Bases: SummaryWriter, *Receiver*

The receiver will be automatically registered to control the Tensorboard.

receive_notify(*solver:* Solver, *message: Dict*)

idrlnet.data module

Define DataNode

class idrlnet.data.**DataNode**(*inputs: Union[Tuple[str, ...], List[str]]*, *outputs: Union[Tuple[str, ...], List[str]]*, *sample_fn: Callable*, *loss_fn: str = 'square'*, *lambda_outputs: Optional[Union[Tuple[str, ...], List[str]]] = None*, *name=None*, *sigma=1.0*, *var_sigma=False*, **args*, ***kwargs*)

Bases: *Node*

A class inherits node.Node. With sampling methods implemented, the instance will generate sample points.

Parameters

- **inputs** (*Union[Tuple[str, ...], List[str]]*) – input keys in return.
- **outputs** (*Union[Tuple[str, ...], List[str]]*) – output keys in return.
- **sample_fn** (*Callable*) – Callable instances for sampling. Implementation of SampleDomain is suggested for this arg.
- **loss_fn** (*str*) – Reduce the difference between a given data and this the output of the node to a simple scalar. square and L1 are implemented currently. defaults to 'square'.
- **lambda_outputs** (*Union[Tuple[str,...], List[str]]*) – Weight for each output in return, defaults to None.
- **name** (*str*) – The name of the node.
- **sigma** (*float*) – The weight for the whole node. defaults to 1.
- **var_sigma** (*bool*) – whether automatical loss balance technique is used. defaults to false
- **args** –
- **kwargs** –

counter = 0

property lambda_outputs

property loss_fn

sample() → *Variables*

Sample a group of points, represented by Variables.

Returns
a group of points.

Return type
Variables

property sample_fn

property sigma

A weight for the domain.

class idrlnet.data.**SampleDomain**

Bases: object

The Template for Callable sampling functions.

abstract sampling(**args*, ***kwargs*)

The method returns sampling points

idrlnet.data.**datanode**(*_fun: Optional[Callable] = None*, *name=None*, *loss_fn='square'*, *sigma=1.0*, *var_sigma=False*, ***kwargs*)

As an alternative, decorate Callable classes as Datanode.

idrlnet.data.**get_data_node**(*fun: Callable*, *name=None*, *loss_fn='square'*, *sigma=1.0*, *var_sigma=False*, **args*, ***kwargs*) → *DataNode*

Construct a datanode from sampling functions.

Parameters

- **fun** (*Callable*) – Each call of the Callable object should return a sampling dict.
- **name** (*str*) – name of the generated Datanode, defaults to None
- **loss_fn** (*str*) – Specify a loss function for the data node.
- **args** –
- **kwargs** –

Returns
An instance of Datanode

Return type
DataNode

idrlnet.data.**get_data_nodes**(*funs: List[Callable]*, **args*, ***kwargs*) → Tuple[*DataNode*]

idrlnet.graph module

Define Computational graph

class idrlnet.graph.**Vertex**(*pre=None*, *next=None*, *node=None*, *ntype='c'*)

Bases: *Node*

counter = 0

class idrlnet.graph.**VertexTaskPipeline**(*nodes: [List[Union[*idrlnet.pde.PdeNode*,* idrlnet.net.NetNode*]]]*, *invar:* Variables, *req_names: List[str]*)

Bases: object

MAX_STACK_ALLOWED = 100000

display(*filename: Optional[str] = None*)

property evaluation_order_list

forward_pipeline(*invar:* Variables, *req_names: Optional[List[str]] = None*) → *Variables*

operation_order(*invar:* Variables)

to_json()

idrlnet.header module

Initialize public objects

class idrlnet.header.**TestFun**(*fun*)

Bases: object

registered = []

static run()

idrlnet.header.**testmemo**(*fun*)

idrlnet.net module

Define NetNode

class idrlnet.net.**NetNode**(*inputs: Union[Tuple, List[str]]*, *outputs: Union[Tuple, List[str]]*, *net: Module*, *fixed: bool = False*, *require_no_grad: bool = False*, *is_reference=False*, *name=None*, **args*, ***kwargs*)

Bases: *Node*

counter = 0

property fixed

property is_reference

load_state_dict(*state_dict: Dict[str, Tensor]*, *strict: bool = True*)

property net

property require_no_grad

state_dict(*destination=None*, *prefix: str = ''*, *keep_vars: bool = False*)

idrlnet.node module

Define Basic Node

class idrlnet.node.**Node**

Bases: object

property derivatives: List[str]

property evaluate: Callable

property inputs: List[str]

property name: str

classmethod new_node(*name: Optional[str] = None*, *tf_eq: Optional[Callable] = None*, *free_symbols: Optional[List[str]] = None*, **args*, ***kwargs*) → *Node*

property outputs: List[str]

idrlnet.optim module

Define Optimizers and LR schedulers

class idrlnet.optim.**Optimizable**

Bases: object

An abstract class for organizing optimization related configuration and operations. The interface is implemented by solver.Solver

OPTIMIZER_MAP = {'ASGD': <class 'torch.optim.asgd.ASGD'>, 'Adadelta': <class 'torch.optim.adadelta.Adadelta'>, 'Adagrad': <class 'torch.optim.adagrad.Adagrad'>, 'Adam': <class 'torch.optim.adam.Adam'>, 'AdamW': <class 'torch.optim.adamw.AdamW'>, 'Adamax': <class 'torch.optim.adamax.Adamax'>, 'LBFGS': <class 'torch.optim.lbfgs.LBFGS'>, 'RMSprop': <class 'torch.optim.rmsprop.RMSprop'>, 'Rprop': <class 'torch.optim.rprop.Rprop'>, 'SGD': <class 'torch.optim.sgd.SGD'>, 'SparseAdam': <class 'torch.optim.sparse_adam.SparseAdam'>}

SCHEDULE_MAP = {'CosineAnnealingLR': <class 'torch.optim.lr_scheduler.CosineAnnealingLR'>, 'CosineAnnealingWarmRestarts': <class 'torch.optim.lr_scheduler.CosineAnnealingWarmRestarts'>, 'CyclicLR': <class 'torch.optim.lr_scheduler.CyclicLR'>, 'ExponentialLR': <class 'torch.optim.lr_scheduler.ExponentialLR'>, 'LambdaLR': <class 'torch.optim.lr_scheduler.LambdaLR'>, 'MultiStepLR': <class 'torch.optim.lr_scheduler.MultiStepLR'>, 'MultiplicativeLR': <class 'torch.optim.lr_scheduler.MultiplicativeLR'>, 'OneCycleLR': <class 'torch.optim.lr_scheduler.OneCycleLR'>, 'StepLR': <class 'torch.optim.lr_scheduler.StepLR'>}

abstract configure_optimizers()

property optimizers

parse_configure(***kwargs*)

parse_lr_schedule(***kwargs*)

parse_optimizer(***kwargs*)

property schedulers

idrlnet.optim.**get_available_class**(*module*, *class_name*) → Dict[str, type]

Search specified subclasses of the given class in module.

Parameters

- **module** (*module*) – The module name
- **class_name** (*type*) – the parent class

Returns
A dict mapping from subclass.name to subclass

Return type
Dict[str, type]

idrlnet.pde module

Define PdeNode

class idrlnet.pde.**ExpressionNode**(*expression*, *name*, ***kwargs*)

Bases: *PdeNode*

class idrlnet.pde.**PdeNode**(*suffix: str = ''*, ***kwargs*)

Bases: *Node*

property equations: Dict

make_nodes() → None

property sub_nodes: List

property suffix: str

idrlnet.receivers module

Concrete predefined callbacks

class idrlnet.receivers.**Notifier**

Bases: object

notify(*obj: object*, *message: Dict*)

property receivers

register_receiver(*receiver:* Receiver)

class idrlnet.receivers.**Receiver**

Bases: object

abstract receive_notify(*obj: object*, *message: Dict*)

class idrlnet.receivers.**Signal**(*value*)

Bases: Enum

An enumeration.

AFTER_COMPUTE_LOSS = 'compute_loss'

BEFORE_BACKWARD = 'signal_before_backward'

BEFORE_COMPUTE_LOSS = 'before_compute_loss'

REGISTER = 'signal_register'

SOLVE_END = 'signal_solve_end'

SOLVE_START = 'signal_solve_start'

```
TRAIN_PIPE_END = 'signal_train_pipe_end'

TRAIN_PIPE_START = 'signal_train_pipe_start'
```

idrlnet.shortcut module

shortcut for API

idrlnet.solver module

Solver

class idrlnet.solver.**Solver**(*sample_domains: Tuple[Union[DataNode, SampleDomain], ...]*, *netnodes: List[NetNode]*, *pdes: Optional[List] = None*, *network_dir: str = './network_dir'*, *summary_dir: Optional[str] = None*, *max_iter: int = 1000*, *save_freq: int = 100*, *print_freq: int = 10*, *loading: bool = True*, *init_network_dirs: Optional[List[str]] = None*, *opt_config: Optional[Dict] = None*, *schedule_config: Optional[Dict] = None*, *result_dir='train_domain/results'*, ***kwargs*)

Bases: *Notifier*, *Optimizable*

Instances of the Solver class integrate configurations and handle the computation operation during solving PINNs. One problem usually needs one instance to solve.

Parameters

- **sample_domains** (*Tuple[DataNode, ...]*) – A tuple of geometry domains used to sample points for training of PINNs.
- **netnodes** (*List[NetNode]*) – A list of neural networks. Trainable computation nodes.
- **pdes** (*Optional[List[PdeNode]]*) – A list of partial differential equations. Similar to net nodes, they can evaluateinputs and output results. But they are not trainable.
- **network_dir** (*str*) – The directory used to automatically load and store ckpt files
- **summary_dir** (*Optional[str]*) – The directory is used for store information about tensorboard. If it is not specified, it will be assigned to network_dir by default.
- **max_iter** (*int*) – Max iteration the solver would run.
- **save_freq** (*int*) – Frequency of saving ckpt.
- **print_freq** (*int*) – Frequency of printing loss.
- **loading** (*bool*) – By default, it is true. It will try to load ckpt and continue previous training stage.
- **init_network_dirs** (*List[str]*) – A list of directories for loading pre-trained networks.
- **opt_config** (*Dict*) – Configure one optimizer for all trainable parameters. It is a wrapper of *torch.optim.Optimizer*. One can specify any subclasses of *torch.optim.Optimizer* by expanding the args like:
  - *opt_config=dict(optimizer='Adam', lr=0.001)* **by default**.
  - *opt_config=dict(optimizer='SGD', lr=0.01, momentum=0.9)*
  - *opt_config=dict(optimizer='SparseAdam', lr=0.001, betas=(0.9, 0.999), eps=1e-08)*

  Note that the opt is Case Sensitive.

- **schedule_config** (*Dict*) – Configure one lr scheduler for the optimizer. It is a wrapper of
  - *torch.optim.lr_scheduler._LRScheduler*. One can specify any subclasses of the class lke:
  - *schedule_config=dict(scheduler='ExponentialLR', gamma=math.pow(0.95, 0.001))*
  - *schedule_config=dict(scheduler='StepLR', step_size=30, gamma=0.1)*

  Note that the scheduler is Case Sensitive.
- **result_dir** (*str*) – save the final training domain data. defaults to 'train_domain/results'
- **kwargs** –

append_sample_domain(*datanode*)

compute_loss(*in_var: Dict[str,* Variables*]*, *pred_out_sample: Dict[str,* Variables*]*, *true_out: Dict[str,* Variables*]*, *lambda_out: Dict[str,* Variables*]*) → Tensor

Compute the total loss in one epoch.

configure_optimizers()

Call interfaces of `Optimizable`

forward_through_all_graph(*invar_dict: Dict[str,* Variables*]*, *req_outvar_dict_index: Dict[str, List[str]]*) → Dict[str, *Variables*]

generate_computation_pipeline()

Generate computation pipeline for all domains. The change of *self.sample_domains* will triger this method.

generate_in_out_dict(*samples: Dict[str,* Variables*]*) → Tuple[Dict[str, *Variables*], Dict[str, *Variables*], Dict[str, *Variables*]]

get_domain_parameter(*domain_name: str*, *parameter: str*)

get_sample_domain(*name: str*) → *DataNode*

infer_step(*domain_attr: Dict[str, List[str]]*) → Dict[str, *Variables*]

Specify a domain and required fields for inference. :param domain_attr: A map from a domain name to the list of required outputs on the domain. :type domain_attr: Dict[str, List[str]] :return: A dict of variables which are required. :rtype: Dict[str, Variables]

init_load()

load()

Load parameters of netnodes and the global step from *model.ckpt*.

property network_dir

property sample_domains

sample_variables_from_domains() → Dict[str, *Variables*]

save()

Save parameters of netnodes and the global step to *model.ckpt*.

set_domain_parameter(*domain_name: str*, *parameter_dict: dict*)

set_param_ranges(*param_ranges: Dict*)

solve()

After the solver instance is initialized, the method could be called to solve the entire problem.

property summary_receiver: *SummaryReceiver*

train_pipe()

Sample once; calculate the loss once; backward propagation once :return: None

property trainable_parameters: List[Parameter]

Return trainable parameters in netnodes. Parameters in netnodes with `is_reference=True` or `fixed=True` will not be returned. :return: A list of trainable parameters. :rtype: List[torch.nn.parameter.Parameter]

idrlnet.torch_util module

conversion utils for sympy expression and torch functions. todo: replace sampling method in GEOMETRY

class idrlnet.torch_util.**integral**(**args*)

Bases: AppliedUndef

default_assumptions = {}

name = 'integral'

idrlnet.torch_util.**torch_lambdify**(*r*, *f*, **args*, ***kwargs*)

idrlnet.variable module

Define variables, intermediate data format for the package.

class idrlnet.variable.**Loss**(*value*)

Bases: Enum

Enumerate loss functions

Identity = 'Identity'

L1 = 'L1'

square = 'square'

class idrlnet.variable.**Variables**

Bases: dict

static cat(**var_list*) → *Variables*

todo: catenate in var list

differentiate_(*independent_var:* Variables, *required_derivatives: List[str]*)

Derivatives will be computed towards the required_derivatives

differentiate_one_step_(*independent_var:* Variables, *required_derivatives: List[str]*)

One order of derivatives will be computed towards the required_derivatives.

classmethod from_tensor(*tensor: Tensor*, *variable_names: List[str]*)

Construct Variables from torch.Tensor

merge_tensor() → Tensor

merge tensors in the Variable

save(*path*, *formats=None*)

Export variable to various formats

subset(*subset_keys: List[str]*) → *Variables*

Construct a new variable with subset references

to_csv(*filename: str*) → None

Export variable to csv

to_dataframe() → DataFrame

merge to a pandas.DataFrame

to_ndarray() → *Variables*[str, np.ndarray]

Return a new numpy based variables

to_ndarray_() → *Variables*[str, np.ndarray]

convert to a numpy based variables

to_torch_tensor_() → *Variables*[str, torch.Tensor]

Convert the variables to torch.Tensor

to_vtu(*filename: str*, *coordinates=None*) → None

Export variable to vtu

static var_differentiate_one_step(*dependent_var:* Variables, *independent_var:* Variables, *required_derivatives: List[str]*)

Perform one step of differentiate towards the required_derivatives

weighted_loss(*name: str*, *loss_function: Union[*Loss*, str]*) → *Variables*

Regard the variable as residuals and reduce to a weighted_loss.

idrlnet.variable.**export_var**(*domain_var: Dict[str,* Variables*]*, *path='./inference_domain/results'*, *formats=None*)

Export a dict of variables to `csv`, `vtu` or `npz`.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

INDEX

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X