Physics-Informed Neural Networks (PINN)

Numerical Analysis for Machine Learning Project

Carlo Sgaravatti (10660072)

Politecnico di Milano
A.Y. 2022/2023

25 January 2023

POLITECNICO
MILANO 1863

Contents

1 Introduction **3**
1.1 Document structure 3

2 PINN's algorithm **3**
2.1 Inverse problems 4

3 Improving PINN's efficiency **4**
3.1 Hard constraints 4
3.2 Residual-based adaptive refinement 5
3.3 Gradient-enhanced PINNs 5

4 Experimental results **5**
4.1 Transport equation with hard constraints 5
4.2 Inverse problem for the transport equation 9
4.3 Comparison of different techniques with the Allen-Cahn equation 12
4.4 Comparison of different techniques with the heat equation 18

1 Introduction

Several numerical approaches have been proposed in order to approximate the solution of the partial differential equations (PDEs). This report aims to show how neural networks can be applied in solving PDEs, using the DeepXDE Python library ([3], [2]). Indeed, the Universal Approximation Theorem states that a Feedforward Neural Network (FNN) can approximate any Borel measurable function from one finite dimensional space to another with any desired non-zero error, provided that the network has enough hidden units and an appropriate activation function (e.g. the sigmoid or the ReLU functions). The key point in this process is to understand how to build an appropriate cost function in order to train the FNN. The Physics-Informed Neural Networks provide an architecture to solve the aforementioned problem by minimizing the PDE residual.

1.1 Document structure

While section 2 describes the PINN's architecture to solve both forward problems and inverse problems, section 3 describes some techniques that can be used in order to improve the efficiency of PINNs. Finally, section 4 presents the experimental results with different examples, using the techniques described in sections 2 and 3.

2 PINN's algorithm

Let's consider the following PDE parametrized by $\boldsymbol{\lambda}$, where $\boldsymbol{x} = (x_1, x_2, \dots, x_d)^T \in \mathbb{R}^d$ and $u(\boldsymbol{x})$ is the solution, defined on a domain $\Omega \subset \mathbb{R}^d$

$$f(\boldsymbol{x}; \frac{\partial u}{\partial x_1}, \dots, \frac{\partial u}{\partial x_d}, \frac{\partial^2 u}{\partial x_1 \partial x_1}, \dots, \frac{\partial^2 u}{\partial x_1 \partial x_d}; \dots; \boldsymbol{\lambda}) = 0, \ \boldsymbol{x} \in \Omega$$

with boundary conditions

$$\boldsymbol{B}(u, \boldsymbol{x}) = 0, \ \boldsymbol{x} \in \partial\Omega$$

The input $\boldsymbol{x}$ can contain the time dimension, so the initial conditions are treated as a special type of boundary conditions. The PINN's algorithm constructs a neural network $\hat{u}(\boldsymbol{x}; \boldsymbol{\theta})$, with parameters $\boldsymbol{\theta}$, which takes $\boldsymbol{x}$ as inputs and will be used to approximate the function $u(\boldsymbol{x})$. In order to train the neural network, the algorithm uses the residual of the PDE and of the boundary conditions for building the loss function. For the training process a set of points $\mathcal{T} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x}_{|\boldsymbol{\mathcal{T}}|}\} \subset \Omega$ is sampled from the domain. In particular, the set of points $\mathcal{T}$ contains both the training points for the PDE, $\mathcal{T}_f \subset \Omega$, and the set of training points for the boundary conditions, $\mathcal{T}_b \subset \partial\Omega$, i.e. $\mathcal{T} = \mathcal{T}_f \cup \mathcal{T}_b$. From these sets of points, the loss function is built in the following way:

$$\mathcal{L}(\boldsymbol{\theta}, \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}, \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}, \mathcal{T}_b)$$

where

$$\mathcal{L}_f(\boldsymbol{\theta}, \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{\boldsymbol{x} \in \mathcal{T}_f} \left\| f(\boldsymbol{x}; \frac{\partial \hat{u}}{\partial x_1}, \dots, \frac{\partial \hat{u}}{\partial x_d}, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \dots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \dots; \boldsymbol{\lambda}) \right\|_2^2$$

$$\mathcal{L}_b(\boldsymbol{\theta}, \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{\boldsymbol{x} \in \mathcal{T}_b} \|\boldsymbol{B}(\hat{u}, \boldsymbol{x})\|_2^2$$

The two parameters w_f and w_b are used to give different weights to the residuals of the PDE and of the boundary conditions, respectively. For example, if the boundary conditions are more critical w.r.t. the PDE, then it is possible to choose $w_b > w_f$.

2.1 Inverse problems

Inverse problems are characterized by the presence of an unknown parameter $\boldsymbol{\lambda}$ in the PDE, for example, an unknown constant or function. In this case, some information (usually, some observation of the real value of u) about the target function u in some points of the domain need to be available. To handle this problem, an additional term is added to the loss function: if $\mathcal{T}_i \subset \Omega$ is the set of points previously described and

$$\mathcal{I}(\boldsymbol{x}, u) = 0, \ \boldsymbol{x} \in \mathcal{T}_i$$

is the relation between u and these points, then the loss function can be modified in the following way

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_b) + w_i \mathcal{L}_i(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_i)$$

where

$$\mathcal{L}_i(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_i) = \frac{1}{|\mathcal{T}_i|} \sum_{\boldsymbol{x} \in \mathcal{T}_i} \|\mathcal{I}(\boldsymbol{x}, \hat{u})\|_2^2$$

3 Improving PINN's efficiency

3.1 Hard constraints

In some cases, it is possible to simplify the learning process by forcing the output of the neural network to satisfy the boundary conditions, that are called hard constraints ([4]). In particular, this is possible when it is easy to represent the boundary (e.g for some kind of Dirichlet or Robin boundary conditions). Considering the following Dirichlet boundary condition

$$u(\boldsymbol{x}) = g_D(\boldsymbol{x}), \ \boldsymbol{x} \in \partial\Omega_D$$

where $\partial\Omega_D \subseteq \partial\Omega$ is a subset of the boundary of the domain of the PDE. Let $b_D(\boldsymbol{x})$ be a function with the following characteristics:

- $b_D(\boldsymbol{x}) = 0$ if and only if $\boldsymbol{x} \in \partial\Omega_D$
- $b_D(\boldsymbol{x}) > 0$ if and only if $\boldsymbol{x} \notin \partial\Omega_D$

Then, it is possible to not consider the previous boundary condition in the training process and directly force the output of the neural network to satisfy it in the following way:

$$\hat{u}(\boldsymbol{x}; \theta) = b_D(\boldsymbol{x}) u_{NN}(\boldsymbol{x}; \theta) + g_D(\boldsymbol{x})$$

where u_{NN} is the output of the neural network.

3.2 Residual-based adaptive refinement

The solution of some PDEs may present some areas where the solution is steep or present a discontinuity. Depending on the training points $\mathcal{T}$, the neural network may find some difficulties in capturing these discontinuities. Indeed, if the training points are randomly sampled, it might occur that there are not enough points near this area. In this case, the Residual-based Adaptive Refinement (RAR, [3]) method can be used to improve the solution obtained. The idea of RAR is to add some points where the PDE residual is higher and then train the neural network another time with the new set of points. In particular, the algorithm performs the following steps:

1. Train the neural network using an initial training set $\mathcal{T}_0$
2. Estimate the mean PDE residual $\mathcal{E}$ using some randomly sampled points $\mathcal{S} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_{|\mathcal{S}|}}\}$:

$$\mathcal{E} = \frac{1}{|\mathcal{S}|} \sum_{\boldsymbol{x} \in \mathcal{S}} \left\| f(\boldsymbol{x}; \frac{\partial \hat{u}}{\partial x_1}, \dots, \frac{\partial \hat{u}}{\partial x_d}, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \dots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \dots; \boldsymbol{\lambda}) \right\|$$

3. If $\mathcal{E}$ is lower than a certain threshold $\mathcal{E}_{th}$ then stop. Otherwise, select the first m points of $\mathcal{S}$ with the highest residual, $\mathcal{S}_m$ and update $\mathcal{T}_k = \mathcal{T}_{k-1} \cup \mathcal{S}_m$; then retrain the network using the set of training points $\mathcal{T}_k$ and return to the previous step.

3.3 Gradient-enhanced PINNs

In the classic PINN architecture, only the PDE residuals are considered. However, since these residuals (represented by the f function) are zero, also the gradient of f is zero. Gradient-enhanced PINNs (gPINNs, [5]) add the gradient information to the loss function by setting all the partial derivatives of f (w.r.t to the input $\boldsymbol{x} \in \Omega$) to zero. Considering a general inverse problem, the new loss function is the following:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_b) + w_i \mathcal{L}_i(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_i) + \sum_{i=1}^{d} w_{g_i} \mathcal{L}_{g_i}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T})$$

where

$$\mathcal{L}_{g_i}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}) = \frac{1}{|\mathcal{T}|} \sum_{\boldsymbol{x} \in \mathcal{T}} \left| \frac{\partial f}{\partial x_i}(\boldsymbol{x}) \right|^2$$

4 Experimental results

4.1 Transport equation with hard constraints

For this example I have considered the following PDE:

$$\begin{cases} \frac{\partial u}{\partial t} + 3\frac{\partial u}{\partial x} = tx, \ x \in \mathbb{R}, \ t \geq 0 \\ u(x, 0) = e^{-x^2}, \ x \in \mathbb{R} \end{cases}$$

This has the following solution:

$$u(x,t) = e^{(x-3t)^2} + \frac{t^2}{2}(x-t)$$

that has the following shape:

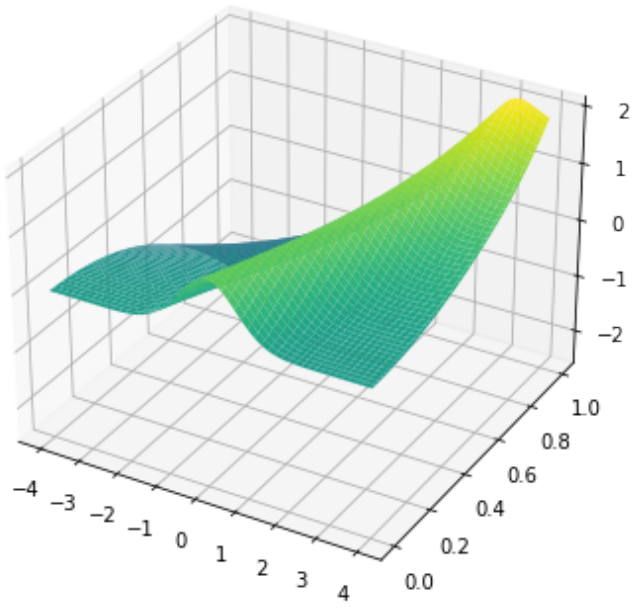


Figure 1: The solution of the PDE

The PDE and the initial condition are represented by the following two functions in python:

```
def pde(x, u):
  du_t = dde.gradients.jacobian(u, x, i=0, j=1)
  du_x = dde.gradients.jacobian(u, x, i=0, j=0)
  return du_t + 3 * du_x - x[:, 1:2] * x[:, 0:1]

def initial_condition(x):
  return np.exp(-(x[:, 0:1])**2)
```

I have solved this problem in two ways:

1. Using a classic PINN, with no hard constraints:

```
geom = dde.geometry.Interval(-4, 4)
time = dde.geometry.timedomain.TimeDomain(0, 1)
geomtime = dde.geometry.GeometryXTime(geom, time)
```

```
bc = dde.icbc.IC(geomtime, initial_condition,
                 lambda x, on_initial: np.isclose(x[1], 0))
data = dde.data.TimePDE(geomtime, pde, [bc], num_domain=1000,
                        num_boundary=100, num_initial=200, num_test=320)

net = dde.maps.FNN([2] + [50] * 3 + [1], 'tanh', 'Glorot uniform')

model = dde.Model(data, net)

model.compile('adam', lr=0.001)
model.train(iterations=20000)
model.compile('L-BFGS')
loss_history, train_state = model.train()
dde.saveplot(loss_history, train_state, issave=True, isplot=True)
```

2. Using a hard constraint for the initial condition, by applying the following transformation to the output of the neural network:

$$\hat{u}(x,t) = tu_{NN}(x,t) + e^{-x^2}$$

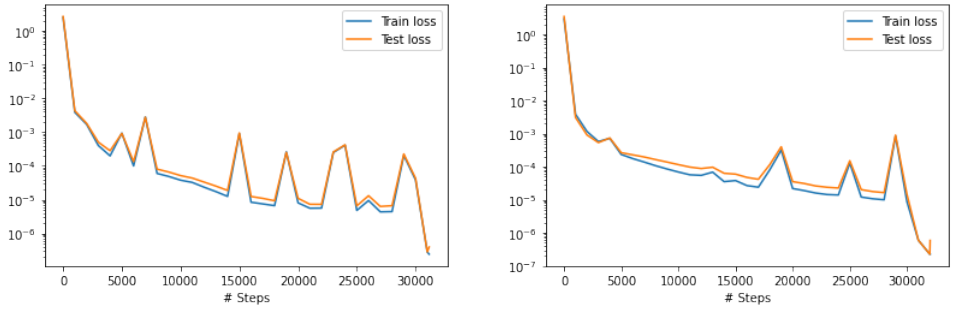
In this way, the boundary constraints are imposed since when $t = 0$ the output is exactly $\hat{u}(x,0) = e^{-x^2}$. In python, this is done with the following lines of code:

```
data = dde.data.TimePDE(geomtime, pde, [], num_domain=1000,
                        num_boundary=100, num_initial=200, num_test=320)
net = dde.maps.FNN([2] + [50] * 3 + [1], 'tanh', 'Glorot uniform')

def transform(x, y):
  t = x[:, 1:2]
  x_ = x[:, 0:1]
  return t * y + tf.exp(-(x_**2))

net.apply_output_transform(transform)
model_hard = dde.Model(data, net)
```

It is possible to notice that the initial condition is no anymore present in the definition of the data, since it is no anymore needed because it is imposed by the output transformation.

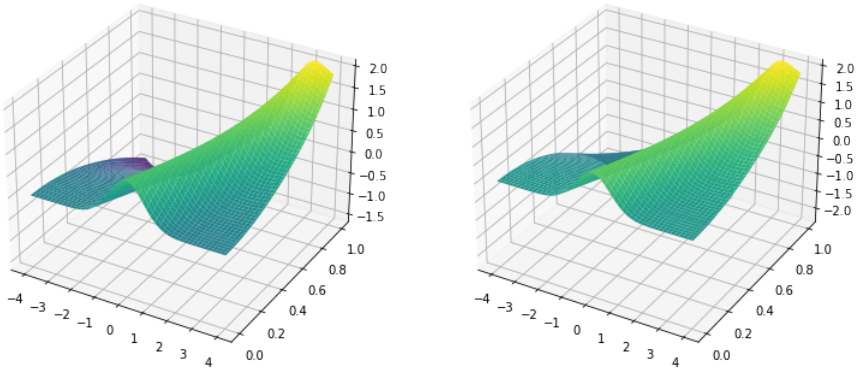
Both cases have the same amount of training points: 1000 points inside the domain, 100 in the boundary and 200 for the initial condition. The two models have also the same neural network architecture (an FNN with 3 hidden layers of 50 neurons) and were trained in the same way: 20000 iterations using the Adam optimizer, using a constant learning rate $\gamma = 0.001$, and some iterations of the L-BFGS optimizer to archive a smaller final loss.

Figure 2 shows the history of the loss function in the training process and Figure 3 shows the corresponding solution. It is possible to notice that the loss and the solution are very similar in the two cases; however, this was not the main objective of the analysis for this experiment.



(a) Without hard constraints (b) With hard constraints

Figure 2: The history of the total loss.



(a) Without hard constraints (b) With hard constraints

Figure 3: The history of the total loss.

Using only these plots it seems that the hard constraints are not improving a lot the solution. However, Figure 4 shows that the predictions are much more accurate in the case of hard constraints. Indeed, in the situation without hard constraints, the maximum reached squared error is 1.2, while in the situation with hard constraints, it is 0.035. This is thanks to the fact that the initial condition is exact in the second case, while in the first case, it has some small errors. Indeed, the second plot shows that the prediction with hard constraints is able to represent correctly the shape of the function even in the top-left corner of the domain, while the PINN with soft constraints was not able to do.

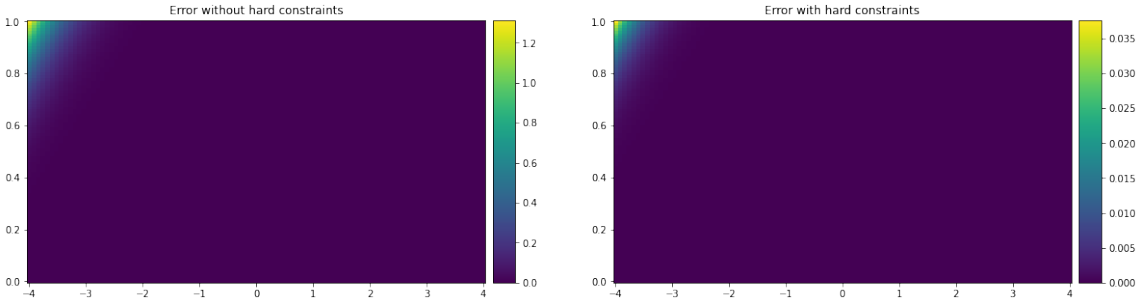


Figure 4: The errors of the predictions

4.2 Inverse problem for the transport equation

As an example of an inverse problem, I considered the following PDE:

$$\begin{cases} \frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = q(x,t), \ x \in \mathbb{R}, \ t \geq 0 \\ u(x,0) = 0, \ x \in \mathbb{R} \end{cases}$$

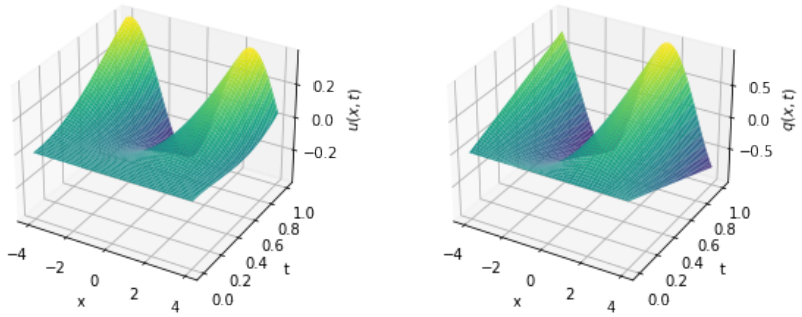
where the function q is considered unknown and the purpose of this example is to find both u and q. The reference value of q is:

$$q(x,t) = t\sin(x)$$

that is associated with the following solution for u:

$$u(x,t) = -\frac{t}{v}\cos x + \frac{1}{v^2}(\sin(x) - \sin(x - vt))$$

These two functions have the following shapes:



(a) (b)

Figure 5: The true values of u and q.

The following lines of code represent the specified problem:

```
def pde(x, y):
  u, q = y[:, 0:1], y[:, 1:2]
  du_t = dde.gradients.jacobian(y, x, i=0, j=1)
  du_x = dde.gradients.jacobian(y, x, i=0, j=0)
  return du_t + v * du_x - q

def initial_condition(x):
  return np.zeros(x.shape[0])

def boundary_initial(x, on_initial):
  return on_initial and np.isclose(x[1], 0)
```

It is possible to notice that the output of the neural network is 2-dimensional because both u and q are being learned. In order to solve the problem, some observations of the true values of u are needed. These have been generated in the following way:

```
def exact_q(x):
    return x[:, 1:2] * np.sin(x[:, 0:1])

def exact_u(x):
  return - (x[:, 1:2] / v) * np.cos(x[:, 0:1]) +
            ((np.sin(x[:, 0:1]) - np.sin(x[:, 0:1] - v * x[:, 1:2])) / (v**2))

def generate_observations(num_points):
  x = np.linspace(-L, L, num_points)
  t = np.linspace(0, T, num_points)
  x, t = np.meshgrid(x, t)
  inputs = np.column_stack((x.flatten(), t.flatten()))
  return inputs, exact_u(inputs)
```

For the training process, 900 points have been generated using the previous function. The domain have been restricted to $x \in [-L, L]$ and $t \in [0, T]$, where $L = 4$ and $T = 1$.

```
geom = dde.geometry.Interval(-L, L)
time = dde.geometry.timedomain.TimeDomain(0, T)
geomtime = dde.geometry.GeometryXTime(geom, time)

ic = dde.icbc.IC(geomtime, initial_condition, boundary_initial)
obs_in, obs_out = generate_observations(30) # 900 points
obs = dde.icbc.PointSetBC(obs_in, obs_out, component=0)

data = dde.data.TimePDE(geomtime, pde, [ic, obs], num_domain=5,
                        num_boundary=5, num_initial=20,
                        anchors=obs_in, num_test=1152)
activation = "tanh"
initializer = "Glorot uniform"
net = dde.nn.PFNN([2] + [[32, 32]] * 3 + [2], activation,
```

```
                initializer, regularization="l2")

weights = [1, 1, 10]
model = dde.Model(data, net)
model.compile("adam", lr=0.001, loss_weights=weights)
losshistory, train_state = model.train(iterations=20000)
model.compile("L-BFGS", loss_weights=weights)
losshistory, train_state = model.train()
dde.saveplot(losshistory, train_state, issave=True, isplot=True)
```

A parallel feedforward neural network (PFNN) has been used instead of the FNN. The PFNN contains two fully connected networks with 3 layers of 32 neurons. The model has been trained using the Adam optimizer for 20000 iterations and using the L-BFGS optimizer in order to archive a better loss, starting from a point closer to the minimum w.r.t the initial guess (thanks to the previous iterations of Adam). It is possible to notice that the weights are not all equal, indeed both the PDE and the initial condition have weights $w_f = 1$ and $w_b = 1$, while the observations have $w_i = 10$. In this way, the contribution of the observations is higher than the other two.

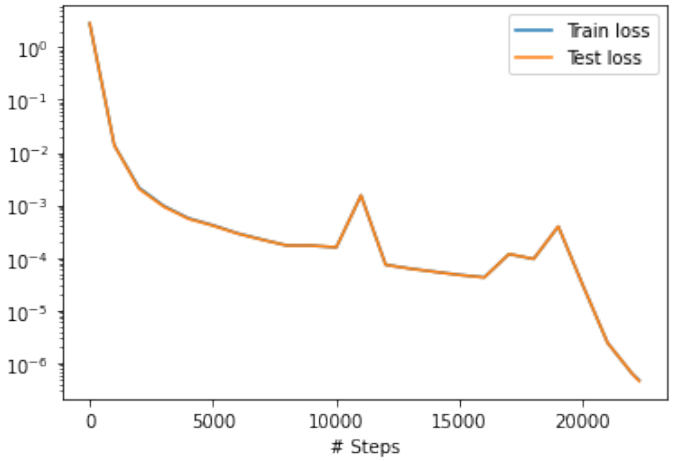
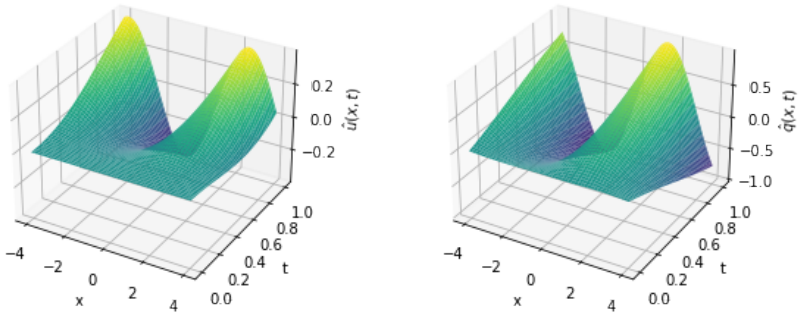


Figure 6: The history of the loss.

In Figure 7 it is shown both the prediction of the solution of the PDE and of the unknown function. It is possible to notice that both functions are quite close to the true ones. Indeed, Figure 8 shows the error of the prediction, which is very low for both cases: in the PDE solution, the error is always lower than 10^{-6} and in the unknown function q the error is always lower than 0.0004 and usually is lower than 10^{-4}.



(a) (b)

Figure 7: The predicted values of u and q.

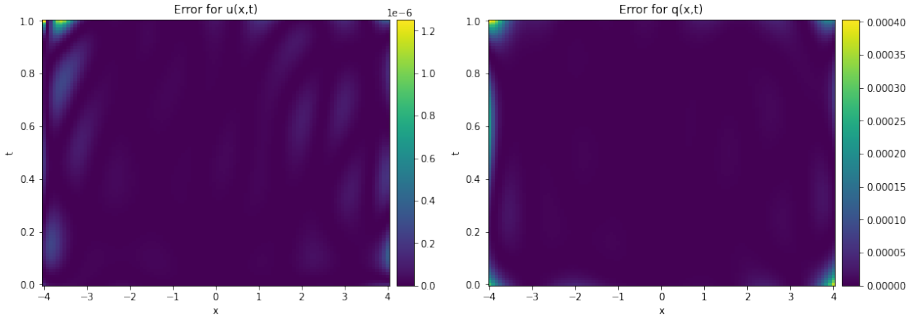


Figure 8: The error of the prediction.

4.3 Comparison of different techniques with the Allen-Cahn equation

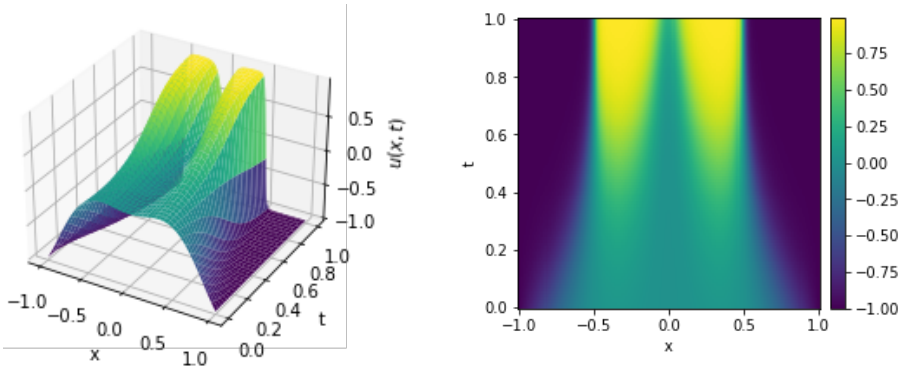
In this example, I wanted to compare different techniques for the following Allen-Cahn equation ([1]):

$$\begin{cases} \frac{\partial u}{\partial t} = d\frac{\partial^2 u}{\partial x^2} + 5(u - u^3), \ x \in [-1,1], \ t \in [0,1] \\ u(x,0) = x^2\cos(\pi x), \ x \in [-1,1] \\ u(-1,t) = u(1,t) = -1, \ t \in [0,1] \end{cases}$$

The boundary conditions and the initial conditions were treated as hard constraints by applying the following transformation to the output:

$$\hat{u}(x,t) = t(1-x^2)u_{NN}(x,t) + x^2\cos(\pi x)$$

Indeed, when $x \in \{0,1\}$ than the output is correctly -1 since $\cos(\pi) = \cos(-\pi) = -1$. In order to check the solution of this equation, i used the dataset present in [1]. Figure 9 shows the shape of the solution, obtained thanks to the dataset previously considered.



(a) 3d solution

(b) 2d projection of the solution

Figure 9: The solution of the Allen-Cahn PDE.

I have considered both PINNs and gPINNs; for the last one, the following two equations, that correspond to set to zero the gradient of f, where considered:

$$\begin{cases} \frac{\partial f}{\partial x} = \frac{\partial^2 u}{\partial t \partial x} - d\frac{\partial^3 u}{\partial x^3} + 5(\frac{\partial u}{\partial x} - 3u^2\frac{\partial u}{\partial x}) = 0 \\ \frac{\partial f}{\partial t} = \frac{\partial^2 u}{\partial t^2} - d\frac{\partial^3 u}{\partial x^2 \partial t} + 5(\frac{\partial u}{\partial t} - 3u^2\frac{\partial u}{\partial t}) = 0 \end{cases}$$

The model for PINN corresponds to:

```
def pde(x, y):
    dy_t = dde.grad.jacobian(y, x, j=1)
    dy_xx = dde.grad.hessian(y, x, i=0, j=0)
    return dy_t - d * dy_xx - 5 * (y - y**3)

def output_transform(x, y):
    return x[:, 0:1]**2 * tf.cos(np.pi * x[:, 0:1]) +
           x[:, 1:2] * (1 - x[:, 0:1]**2) * y
```

while for gPINNs corresponds to:

```
def pde_gpinn(x, y):
    dy_t = dde.grad.jacobian(y, x, i=0, j=1)
    dy_xx = dde.grad.hessian(y, x, i=0, j=0)

    dy_tt = dde.grad.hessian(y, x, i=1, j=1)
    dy_xxt = dde.grad.jacobian(dy_xx, x, i=0, j=1)

    dy_tx = dde.grad.hessian(y, x, i=1, j=0)
    dy_xxx = dde.grad.jacobian(dy_xx, x, i=0, j=0)
    dy_x = dde.grad.jacobian(y, x, i=0, j=0)

    return [
```

```
        dy_t - d * dy_xx - 5 * (y - y**3),
        dy_tx - d * dy_xxx - 5 * (dy_x - 3 * y**2 * dy_x),
        dy_tt - d * dy_xxt - 5 * (dy_t - 3 * y**2 * dy_t)
    ]
```

I have compared the following situations:

1. PINN without RAR or gradient enhancement:

```
def train_pinn(iterations_adam, lr, domain_points, bc_points, ic_points):
    geom = dde.geometry.Interval(-1, 1)
    timedomain = dde.geometry.TimeDomain(0, 1)
    geomtime = dde.geometry.GeometryXTime(geom, timedomain)

    data = dde.data.TimePDE(geomtime, pde, [], num_domain=domain_points,
                            num_boundary=bc_points, num_initial=ic_points)
    net = dde.nn.FNN([2] + [20] * 4 + [1], "tanh", "Glorot normal")
    net.apply_output_transform(output_transform)
    model = dde.Model(data, net)

    model.compile("adam", lr=lr)
    model.train(iterations=iterations_adam)
    model.compile("L-BFGS")
    losshistory, train_state = model.train()
    return losshistory, train_state, model
```

2. 3 iterations of the RAR algorithm applied to a PINN, with a threshold of 0.0001, adding 20 points per iteration to the training set:

```
def train_rar(iterations_adam, lr, domain_points, bc_points, ic_points,
              max_iterations_rar, threshold, m):
    geom = dde.geometry.Interval(-1, 1)
    timedomain = dde.geometry.TimeDomain(0, 1)
    geomtime = dde.geometry.GeometryXTime(geom, timedomain)

    data = dde.data.TimePDE(geomtime, pde, [], num_domain=domain_points,
                            num_boundary=bc_points, num_initial=ic_points)
    net = dde.nn.FNN([2] + [20] * 4 + [1], "tanh", "Glorot normal")
    net.apply_output_transform(output_transform)
    model_rar = dde.Model(data, net)

    model_rar.compile("adam", lr=lr)
    model_rar.train(iterations=iterations_adam)
    model_rar.compile("L-BFGS")
    losshistory, train_state = model_rar.train()

    inputs = geomtime.random_points(10000)
    error = 10
```

```
    i = 0
    added_points = np.empty([0, 2])
    while error > threshold and i < max_iterations_rar:
        f_values = model_rar.predict(inputs, operator=pde)
        f_abs = np.absolute(f_values)
        error = np.mean(f_abs)
        print("mean error: ", error)

        top_m = np.argsort(f_abs.flatten())[::-1][:m]
        points_to_add = inputs[top_m]
        print("points added: ", points_to_add)
        data.add_anchors(points_to_add)
        added_points = np.vstack((added_points, points_to_add))

        model_rar.compile("adam", lr=lr)
        model_rar.train(iterations=iterations_adam, disregard_previous_best=True)
        model_rar.compile("L-BFGS")
        losshistory, train_state = model_rar.train()
        i += 1

    return losshistory, train_state, model_rar, added_points
```

3. gPINN without RAR:

```
def train_gpinn(iterations_adam, lr, domain_points, bc_points,
                ic_points, weights_g):
    geom = dde.geometry.Interval(-1, 1)
    timedomain = dde.geometry.TimeDomain(0, 1)
    geomtime = dde.geometry.GeometryXTime(geom, timedomain)

    data = dde.data.TimePDE(geomtime, pde_gpinn, [], num_domain=domain_points,
                            num_boundary=bc_points, num_initial=ic_points)
    net = dde.nn.FNN([2] + [20] * 4 + [1], "tanh", "Glorot normal")
    net.apply_output_transform(output_transform_gpinn)
    model_gpinn = dde.Model(data, net)

    weights = [1, weights_g, weights_g]
    model_gpinn.compile("adam", lr=lr, loss_weights=weights)
    model_gpinn.train(iterations=iterations_adam)
    model_gpinn.compile("L-BFGS", loss_weights=weights)
    losshistory, train_state = model_gpinn.train()
    return losshistory, train_state, model_gpinn
```

4. 3 iterations of the RAR algorithm applied to a gPINN, with a threshold of 0.0001, adding 20 points per iteration to the training set (the code is not reported since it is a combination of the previous two cases).

All the models consist of 4 layers with 20 neurons each and use the tanh activation function. In

the first analysis done, the models were trained for 30000 epochs of the Adam optimizer and some iterations of the L-BFGS optimizer and the training points were distributed in the following way: 2000 points inside the domain, 200 in the boundary of x and 200 in the boundary of t. The gPINNs had gradients weighted with $w_g = 0.0001$. Figure 10 shows the solution obtained by the PINN without RAR; it is possible to notice that the errors in the PDE residual are quite big, indeed they can reach the value of 0.25, while the maximum error for the solution is in the positions with $x = 0$ and can reach the value of 0.7. Figure 11 shows how RAR have improved the solution. In the first plot, the points that were added during the process are shown. It is possible to notice that these points are indeed the points in which the residual of the PDE in Figure 10 was higher. By adding these points I have archived both a better solution (the maximum error is 0.25) and a smaller residual (the maximum residual is 0.06). Figure 12 shows how the gPINN have built the solution of the PDE. In particular, it is possible to see that the gPINN has improved the solution of the PINN, in particular for the PDE residual that is reduced to a maximum of 0.09 (while the prediction has improved a bit in some areas but the maximum error value is reached when $x = 0$ and is still a bit less than 0.7). Finally, Figure 13 shows how RAR can improve the solution obtained by just having a gPINN or a PINN with RAR. In this case, the residual is significantly reduced to a maximum of 0.025 and the maximum error is a bit more than 0.2 (but the area on which the error is high is significantly reduced).

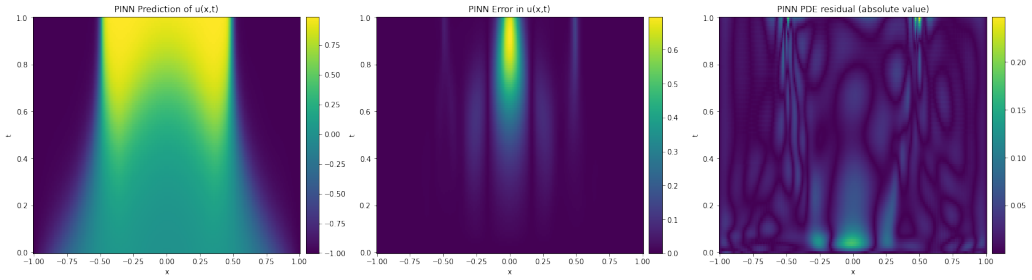


Figure 10: The PINN results.

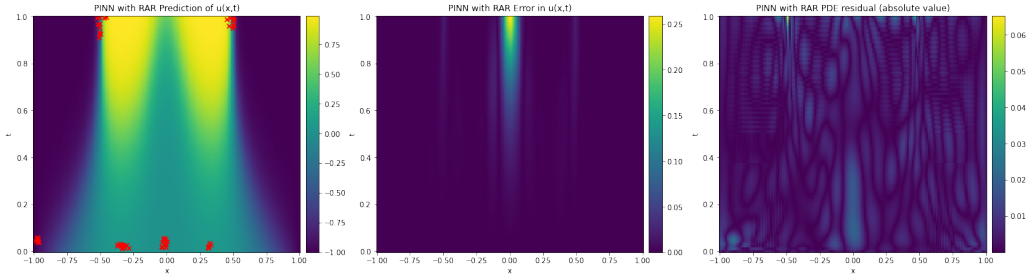


Figure 11: The PINN with RAR results.

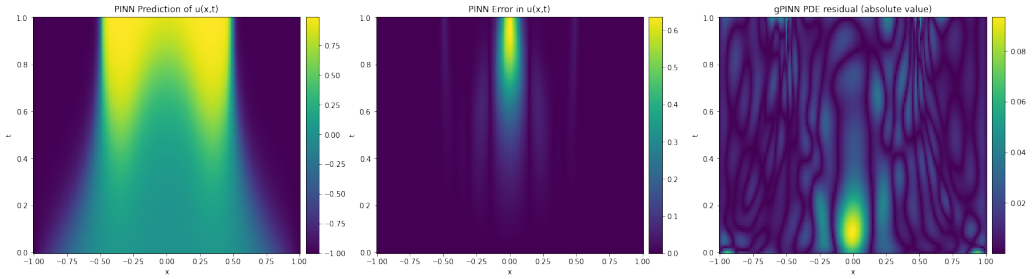


Figure 12: The gPINN results.

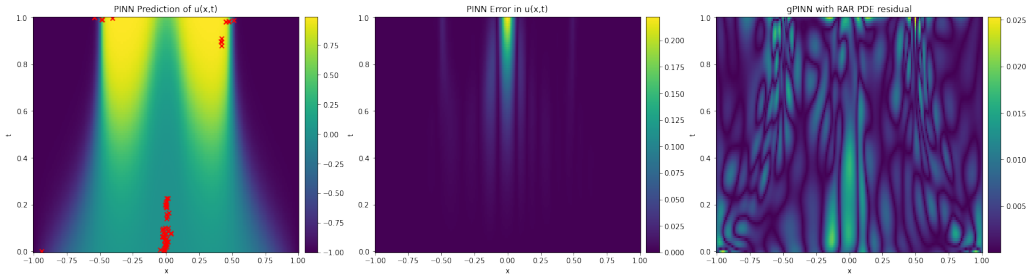


Figure 13: The gPINN with RAR results.

In addition, the models were evaluated using the following metrics:

1. Mean PDE residual:

$$R_{PDE} = \frac{1}{|\mathcal{T}_e|} \sum_{\boldsymbol{x} \in \mathcal{T}_e} \left\| f(\boldsymbol{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}) \right\|$$

   where $\mathcal{T}_e$ is the test set.

2. L2 relative error:

$$L_2 = \frac{\|y - \hat{y}\|_2}{\|y\|_2}$$

   where $y \in \mathbb{R}^n$ is the true value of the outputs (where n is the number of test points) and $\hat{y} \in \mathbb{R}^n$ is the predicted value of the outputs.

```
y_pred = model.predict(X)
f = model.predict(X, operator=pde_gpinn)[0]
print("Mean residual:", np.mean(np.absolute(f)))
print("L2 relative error:", dde.metrics.l2_relative_error(y_true, y_pred))
```

Table 1 shows how the models were performing. One thing that can be noticed is that gPINN is improving the PINN results, while RAR is improving the results in both cases of PINN and gPINN.

	PINN w/o RAR	PINN w/ RAR	gPINN w/o RAR	gPINN w/ RAR
Mean PDE residual	0.0172922	0.0038190	0.0105650	0.0036791
L2 relative error	0.1386853	0.0284568	0.1117112	0.0233802

Table 1: Comparison of the PDE residuals and l2 relative errors

The second type of analysis that I wanted to do was trying to lower both the number of training points and the number of iterations of the Adam optimization process. In particular, I have trained the models described before using 10000 iterations of Adam (plus some iterations of L-BFGS) and selecting 50 points in the boundary of x, 50 points in the boundary of y and different numbers of points inside the domain: 100, 200, 400, 600. Figure 14 shows the PDE mean residual and the l2 relative error obtained with the 4 methods. For the PDE residual, the figure shows how gPINN is able to improve a lot the results, especially when the number of points is really low. In general, it seems that both RAR and gPINN are able to improve a lot the performance of PINNs when the number of points is low. When the number of points is increasing the performance become comparable, however, as it was shown in the first analysis with 2000 points, RAR (with both PINN and gPINN) is able to perform better in areas with steep solutions.

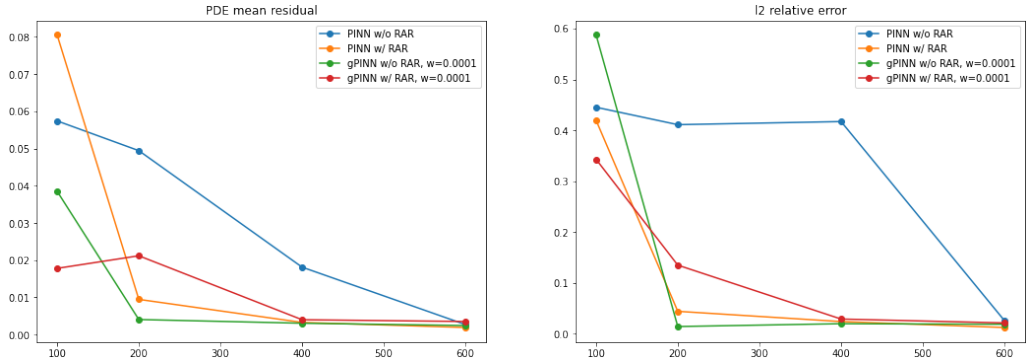


Figure 14: The comparison of the 4 methods with different numbers of training points.

4.4 Comparison of different techniques with the heat equation

In the last example, I have considered the following 1d heat equation with Neumann boundary conditions:

$$\begin{cases} u_t + Du_{xx} = 0, \ x \in [0,1], \ t \in [0,1] \\ u_x(0,t) = u_x(1,t) = 0, \ t \in [0,1] \\ u(x,0) = \sin^4(\pi x), \ x \in [0,1] \end{cases}$$

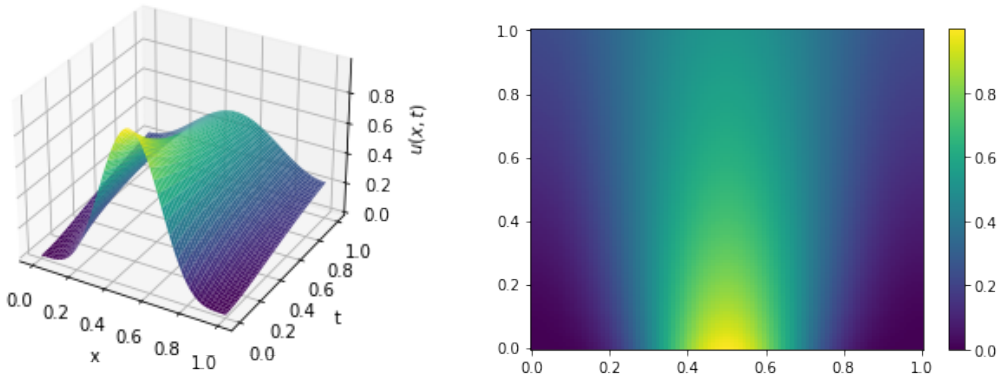
where i have considered $D = 0.03$. This PDE has the following solution:

$$u(x,t) = \frac{3}{8} - \frac{1}{2}e^{-4\pi^2 t}\cos(2\pi x) + \frac{1}{8}e^{-16\pi^2 t}\cos(4\pi x)$$

The PDE is represented in python in this way:

```
def pde(x, u):
    du_t = dde.gradients.jacobian(u, x, i=0, j=1)
    du_xx = dde.gradients.hessian(u, x, i=0, j=0)
    return du_t - D * du_xx

def boundary_bc(x, on_boundary):
    return on_boundary and (np.isclose(x[0], 0) or np.isclose(x[0], 1))

def boundary_ic(x, on_initial):
    return on_initial and np.isclose(x[1], 0)

def boundary_conditon(x):
    return np.zeros((x.shape[0], 1))

def initial_condition(x):
    return np.sin(np.pi * x[:, 0:1])**4
```

Figure 15 shows the shape of the solution.



(a) 3d solution

(b) 2d solution

Figure 15: The solution of the PDE.

The aim of this example is to compare the solution obtained by a PINN with and without points resampling, and then to show how RAR can improve the solution. The points resampling consists of randomly sampling different residual points for the training set during the optimization process after a certain number of iterations. In this case, I have resampled points after every 10 iterations through a callback available in the DeepXDE library:

```
def train_resampler(weights, num_domain, num_boundary, num_initial):
    geo_domain = dde.geometry.geometry_1d.Interval(0, 1)
    time_domain = dde.geometry.timedomain.TimeDomain(0, 5)
    geomtime = dde.geometry.GeometryXTime(geo_domain, time_domain)
```

```
ic = dde.icbc.initial_conditions.IC(geomtime, initial_condition, boundary_ic)
bc = dde.icbc.boundary_conditions.NeumannBC(
        geomtime, boundary_conditon, boundary_bc)
data = dde.data.TimePDE(geomtime, pde, [bc, ic], num_domain=num_domain,
                        num_boundary=num_boundary, num_initial=num_initial)

layer_size = [2] + [32] * 3 + [1]
activation = "tanh"
initializer = "Glorot uniform"
net = dde.nn.FNN(layer_size, activation, initializer)

model = dde.Model(data, net)

pde_resampler = dde.callbacks.PDEPointResampler(period=10)

model.compile("adam", lr=1e-3, loss_weights=weights)
losshistory, train_state = model.train(iterations=30000,
                                       callbacks=[pde_resampler])
model.compile("L-BFGS", loss_weights=weights)
losshistory, train_state = model.train(callbacks=[pde_resampler])
return losshistory, train_state, model
```

Figure 16 shows the results of the training process with a PINN without point resampling, while Figure 17 shows the results of the training process using point resampling with the previous code. In both cases, 200 points have been used for the training set inside the domain, 20 for the boundary condition and 20 for the initial condition. It is possible to see that the situation with points resampling has lowered the residual of the PDE, while the error in the prediction has not improved. In order to improve the error of the prediction, I have used the RAR algorithm, specifying a threshold of 0.0001 for the residuals and a maximum of 5 iterations, adding 10 points at each iteration. Figure 18 shows the results of such a process, where the first plot shows the points that were added during the algorithm (which corresponds indeed to positions where the residual was high in Figure 16). The 2 plot shows that the error in the prediction is still high in some points, however, the central part of the domain is affected by much fewer errors. In addition, the third plot shows that the residual is reduced a lot; indeed here it has a maximum value of around 0.016, while for the PINN solution, it reaches 0.2.

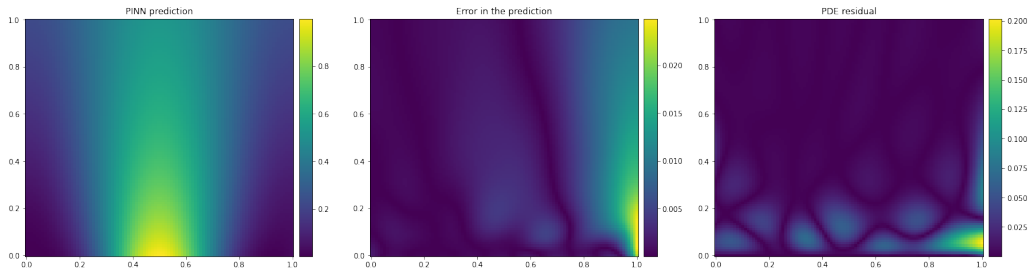


Figure 16: The results of the training process using PINN without RAR or PDE resampling.

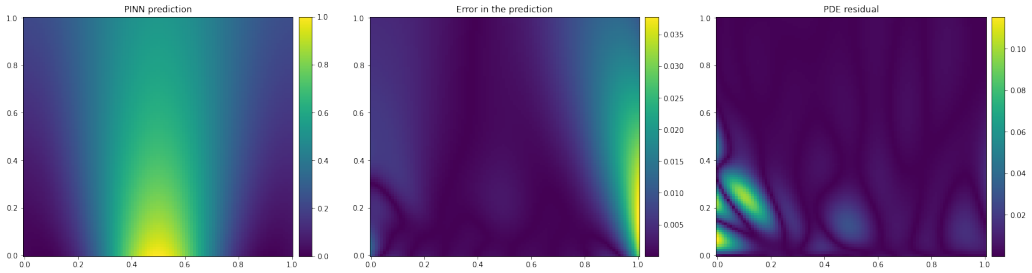


Figure 17: The results of the training process using PINN with PDE resampling.

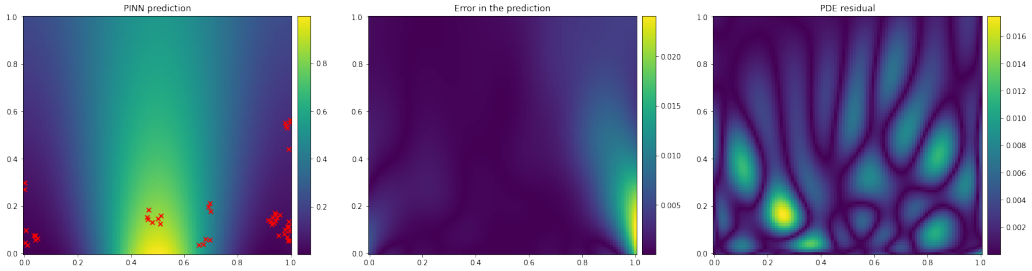


Figure 18: The results of the training process using PINN with RAR.

Finally, I did some tests to compare the performance of PINN with and without points resampling. In particular, I have tested the results with different numbers of the training points inside the domain and using different weights for the boundary and initial conditions in the loss function. The models were evaluated using the mean PDE residual and the l2 relative error. Figure 19 shows the results of such a process: in general, it seems that the situations with points resampling are usually performing slightly better than the situations without points resampling. Regarding the weights of the boundary and initial conditions, having $w = 0.1$ provides, as expected, a lower PDE residual since the PDE counts more than the boundary and the initial conditions for the loss function while having $w = 10$ is associated with a slightly better l2 relative error. For this reason, it is possible to say that the case with $w = 1$ provides a better tradeoff for the two metrics.

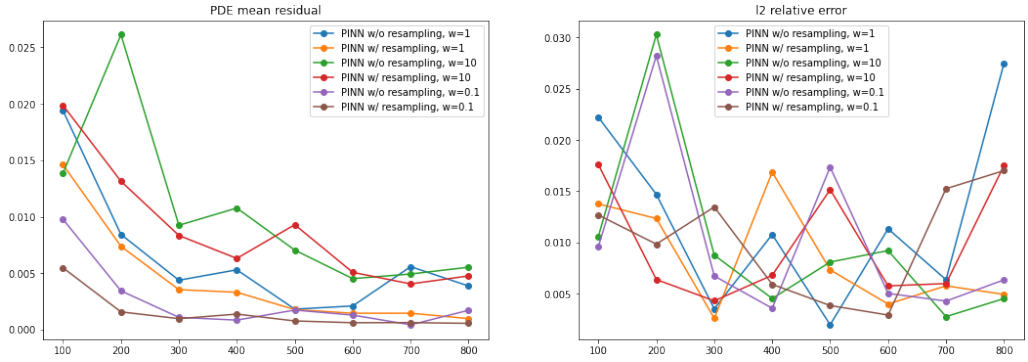


Figure 19: The comparison of the PINN with and without PDE resampling with different numbers of training points and weights of the boundary conditions.

References

[1] Allen-cahn equation. https://deepxde.readthedocs.io/en/latest/demos/pinn_forward/allen.cahn.html. Accessed: 2023-02-23.

[2] Documentation of the deepxde python library. https://deepxde.readthedocs.io/en/latest/index.html.

[3] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. Deepxde: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, 2021.

[4] Lu Lu, Raphaël Pestourie, Wenjie Yao, Zhicheng Wang, Francesc Verdugo, and Steven G. Johnson. Physics-informed neural networks with hard constraints for inverse design. *SIAM Journal on Scientific Computing*, 43(6):B1105–B1132, 2021.

[5] Jeremy Yu, Lu Lu, Xuhui Meng, and George Em Karniadakis. Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems. *CoRR*, abs/2111.02801, 2021.