INRIA - CNRS Alsace delegation

Training augmented interpolation

FONSECA HINCAPIÉ Diana Sol Angel

STAGE M1 CSMI

Supervisors:

M.FRANCK Emmanuel

M.NAVORET Laurent

Contents

1 Objectives

We are interested in solving simple transport equations. The goal of work is to increase the accuracy of the method by using a prediction of the solution learnt using a PINN method. For this we implement the Semi-Lagrangian scheme and a simple case of deep interpolation where we approximate the solution of our transport equation by training a physics informed neural network. We used a machine learning framework like PyTorch to implement PINNs.

2 Introduction

The focus of the internship was on solving simple transport equations using deep learning techniques, specifically Physics-Informed Neural Networks (PINNs).I will start by discussing the importance of solving transport equations numerically and the role of neural networks in this process.

Then, I will provide a theoretical framework for understanding the Semi-Lagrangian scheme with augmented interpolation and PINNs. In the implementation section I will describe the code developed.

In the results section I will present the outcomes of the PINNs strategy including the results of unsupervised and supervised learning approaches and it's implementation in the Semi-Lagrangian scheme.

Finally, I will draw conclusions based on the findings and discuss the implications of the work.

3 General context

The internship project was supervised by Emmanuel Franck and Laurent Navoret researchers on the development of numerical methods to solve partial differential equations for the simulation of physical phenomena from the National Institute for Research in Digital Science and Technology (INRIA) which is a leading research institution focused on computer science, applied mathematics, and control theory and the University team 'Modeling and Control' of the IRMA laboratory. The work is done in collaboration with both institutions.

4 Theoretical Framework

In order to provide a solid conceptual foundation, this section provides the theoretical framework that allows to understand the context and the fundamental concepts of the work carried out.

4.1 Transport equations

We are particularly interested in solving the following transport equation:

$$\begin{cases} \partial_t u + a\partial_x u = 0 \\ u(t=0,x) = u_0(x,\mu) \end{cases}$$

With μ a parameter of the equation, and the velocity of the transport.

This equation has an analytical solution:

$$u(t,x) = u_0(x, x-at)$$

4.2 Lagrange interpolation

We choose n points inside $(x_1, \ldots, x_m)$. The Lagrange interpolation operator is defined as it follows:

$$\mathcal{I}_h^m(f)(x) = \sum_{i=1}^{m} f\left(x_i\right) P_i(x).$$

with $P_i\left(x_j\right) = \delta_{ij}$.

4.3 Semi-Lagrangian Scheme

The semi-Lagrangian scheme is a numerical method to solve transport equations. It is based on the following idea for the 1D case.

First to allow long time-steps, we construct the numerical domain of dependence to contain the physical domain of dependence.

We have an uniform grid for space $[x_1, \dots x_N]$ and one for the time $[t_1, ..t_n]$ with $\Delta t = t_{i+1} - t_i$ and $h = \Delta x = x_{i+1} - x_i$.
The numerical solution can be computed using the following scheme

$$u\left(t_{n+1}, x_j\right) = u\left(t_n, x_j - a\Delta t\right) = u(t_{n+1}, x_*).$$

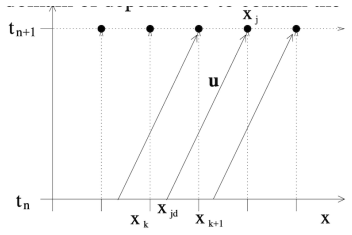


Figure 1: Semi-Lagrangian Scheme

With $x_* = x_j - a\Delta t$ but we can clearly see that our x_* is not a point of our mesh and consequently we cannot know $u\left(t_n, x_i - a\Delta t\right)$. Is at this point that we make use of the following scheme.

$$u\left(t_{n+1}, x_i\right) = \mathcal{I}_h^m\left(u\left(t_n, x_i\right)\right)\left(x_i - a\Delta t\right).$$

And introduce an approximated solution u_j^n where:

$$u_i^{n+1} = \mathcal{I}_h^m((u_j^n)_j)(x_i - a\Delta t)$$

Where we interpolate u at time $n+1$ from the n-closest points at time n and onto x_*, using our Lagrange interpolation operator described before $\mathcal{I}_h^m$ uses the n points around $(x_i - a\Delta t)$.

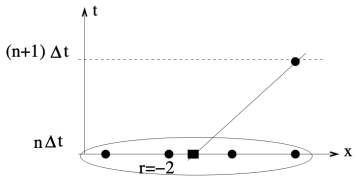


Figure 2: N-closest points for interpolation

Such a procedure is called Semi-Lagrangian because the discrete solution is given on an Eulerian fixed mesh, but on the other hand the scheme relies on the construction of $u\left(t_n, x_j - a\Delta t\right)$ which is used to approximate $u\left(t_{n+1}, x_j\right)$.

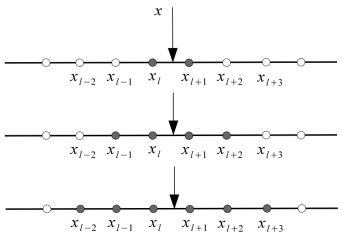


Figure 3: Different interpolation options

4.4 Augmented Interpolation

The deep Lagrange interpolation operator is defined as follows:

$$\mathcal{I}_d^m(f) = \sum_{i=1}^{n} \frac{f(x_i)}{u_{pred}(x_i)} P_i(x) u_{pred}(x).$$

Where u_{pred} is the prediction of u and with $P_i(x_j) = \delta_{ij}$

Using this choice, we obtain that $\mathcal{I}_d(f)(x_i) = f(x_i)$ as the classical Lagrange interpolation operator.

4.4.1 Deep Lagrange interpolation with PINNs

We will consider a specific case of the deep interpolation:

$$\mathcal{I}_d^m(f) = \mathcal{I}^m\left(\frac{f}{u_\theta}\right) u_{pred}(x)$$

This interpolation will be better than the classical one if $\frac{f}{u_\theta} \approx= 1$ since we have the following error on the interpolation:

$$\|f - \mathcal{I}_d^m(f)\|_{H^m} \leq Ch^{m+1} \left\| \left(\frac{f}{u_\theta}\right)'' \right\|_{L^2} \|f\|_{H^m}.$$

Hence, the interpolation error will be smaller if u_θ is good approximation of the true solution

Now the question is how choose $u_\theta(x)$. We propose to use a neural network which will approximate the $u_\theta(x)$ function.

$$u_\theta(x, t, \mu, \sigma, a).$$

We will train these neural networks with a **Physics Informed Neural Network** strategy, and we will use the previous interpolation to approximate the solution of the PDE.

4.5 PINNs for solving PDEs

In this project we were particularly interested in approximating the $u\theta$ function, which is the solution of the following PDE:

Advection Equation

$$\begin{cases} \frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \quad (x,t) \in \Omega \times (0,T) \\ u(x, t=0) = u_0(x) \quad x \in \delta\Omega \end{cases} \tag{1}$$

The problem (1) represents a wave propagating with a constant velocity a with an unchanged shape and has an analytical solution equal to:

$$u(t,x) = u_0(x, x - at)$$

We have an initial conditions u_0 that we will be using:

Gaussian distribution

Defined as it follows:

$$u_0 = \exp\left(-\frac{(x-\mu)^2}{\sigma}\right)$$

with μ the mean and σ the variance of the Gaussian distribution.

4.6 Supervised and unsupervised learning

4.6.1 Supervised learning

Supervised learning is a subcategory of machine learning and artificial intelligence. It is defined by its use of labeled datasets to train algorithms that to classify data or predict outcomes accurately. As input data is fed into the model, it adjusts its weights until the model has been fitted appropriately, which occurs as part of the cross validation process. Supervised learning uses a training set to teach models to yield the desired output. This training dataset includes inputs and correct outputs, which allow the model to learn over time. The algorithm measures its accuracy through the loss function, adjusting until the error has been sufficiently minimized.

4.6.2 Unsupervised learning

Unsupervised learning uses machine learning algorithms to analyze and cluster unlabeled datasets. These algorithms discover hidden patterns or data groupings without the need for human intervention. Its ability to discover similarities and differences in information make it the ideal solution for exploratory data analysis, cross-selling strategies, customer segmentation, and image recognition.

In our context we will be interested in both supervised and unsupervised learning. We will use unsupervised learning to train the neural networks without knowing the solution and supervised learning to learn from the data calculated based on the exact solution of the PDE.

4.7 Neural Networks

Neural networks process training data by mimicking the interconnectivity of the human brain through layers of nodes, each node is made up of inputs, weights, a bias (or threshold), and an output. If that output value exceeds a given threshold, it "fires" or activates the node, passing data to the next layer in the network. Neural networks learn this mapping function through supervised learning, adjusting based on the loss function through the process of gradient descent. When the cost function is at or near zero, we can be confident in the model's accuracy to yield the correct answer.

They are basically functions that map an input X to an output Y by performing successive linear and nonlinear transformations. The linear transformations are represented by a set of weights W and biases b and the nonlinear transformations are represented by activation functions σ. The output of a neural network is given by:

$$\overline{u_\theta}(X) = W_n\sigma_{n-1}(W_{n-1}\sigma_{n-2}(...(W_2(W_1X + b_1) + b_2) + ..) + b_{n-1}) + b_n$$

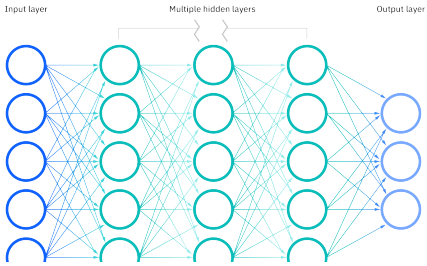


Figure 4: Neural Network Architecture — **Source:** IBM Cloud Education

4.7.1 Multi Layer Perceptron (MLP)

A Multilayer Perceptron has input and output layers, and one or more hidden layers with many neurons stacked together. And while in the Perceptron the neuron must have an activation function that imposes a threshold, like ReLU or sigmoid, neurons in a Multilayer Perceptron can use any arbitrary activation function. Multilayer Perceptron falls under the category of feedforward algorithms, because inputs are combined with the initial weights in a weighted sum and subjected to the activation function, just like in the Perceptron. But the difference is that each linear combination is propagated to the next layer. Each

layer is feeding the next one with the result of their computation, their internal representation of the data. This goes all the way through the hidden layers to the output layer. But it has more to it. If the algorithm only computed the weighted sums in each neuron, propagated results to the output layer, and stopped there, it wouldn't be able to learn the weights that minimize the cost function. If the algorithm only computed one iteration, there would be no actual learning.

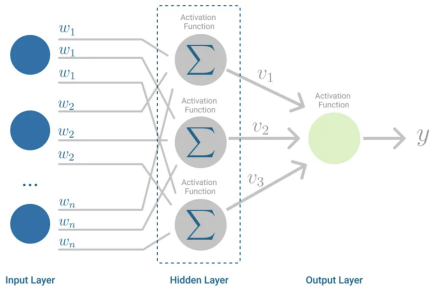


Figure 5: Multi Layer Perceptron — **Source:** Multilayer Perceptron Explained with Real-Life Example and Python

4.7.2 Backpropagation

Backpropagation is a learning mechanism that allows the Multilayer Perceptron to iteratively adjust the weights in the network, with the goal of minimizing the cost function. There is one hard requirement for backpropagation to work properly. The function that combines inputs and weights in a neuron, for instance the weighted sum, and the threshold function, for instance ReLU, must be differentiable. These functions must have a bounded derivative, because Gradient Descent is typically the optimization function used in MultiLayer Perceptron. In each iteration, after the weighted sums are forwarded through all layers, the gradient of the Mean Squared Error is computed across all input and output pairs. Then, to propagate it back, the weights of the first hidden layer are updated with the value of the gradient. That's how the weights are propagated back to the starting point of the neural network

Then we get the following infographic that shows the whole process of a Multilayer Perceptron with feedforward and backpropagation:

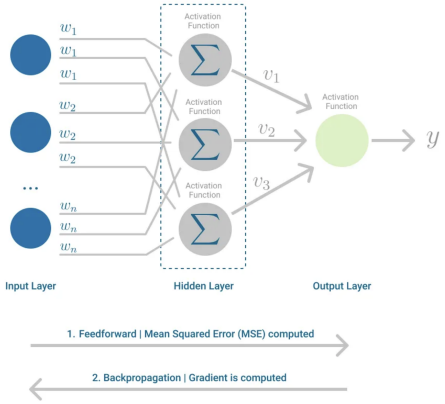


Figure 6: Multi Layer Perceptron and Backpropagation - — **Source:** Multilayer Perceptron Explained with Real-Life Example and Python

4.7.3 Activation functions

The activation function that we choose for out neural network has an impact or its training. There are many commonly used activation functions like Sigmoid, Tanh, ReLU, we usually use the infinitely differentiable hyperbolic tangent activation function $tanh(x)$ that is perfect for PINN. The regularity of PINNs can be ensured by using smooth activation functions like as the sigmoid and hyperbolic tangent, allowing estimations of PINN generalization error to hold true

4.7.4 Universal Approximation Theorem

The representational ability of neural networks is well established. According to the universal approximation theorem: The universal approximation theorem states that any continuous function

$$f : [0, 1]^n \to [0, 1]$$

can be approximated arbitrarily well by a neural network with at least 1 hidden layer with a finite number of weights. Even if neural networks can express very complex functions compactly, determining the precise parameters (weights and biases) required to solve a specific PDE can be difficult.

4.8 Physics Informed Neural Networks

Physics-informed neural networks (PINNs) are a type of universal function approximators that can embed the knowledge of any physical laws that govern a given data-set in the learning process, and can be described by partial differential equations (PDEs).

They approximate PDE solutions by training a neural network to minimize a loss function; it includes terms reflecting the initial and boundary conditions along the space-time domain's boundary and the PDE residual at selected points in the domain (called collocation point). They are deep-learning networks that, given an input point in the integration domain, produce an estimated solution in that point of a differential equation after training. Incorporating a residual network that encodes the governing physics equations is a significant novelty with PINNs.

The basic concept behind PINN training is that it can be thought of as an unsupervised strategy that does not require labelled data, such as results from prior simulations or experiments. It works by integrating the mathematical model into the network and reinforcing the loss function with a residual term from the governing equation, which acts as a penalizing term to restrict the space of acceptable solutions.

4.8.1 The Loss Function

Using the PINNs approach we determine the parameters θ of the NN, u_θ, by minimizing the loss function:

$$\theta = argmin_\theta L(\theta)$$

Where the loss function is defined as:

$$L(\theta) = L_{data}(\theta) + L_{physics}(\theta) + L_{SBC}(\theta) + L_{TBC}(\theta) + L_{IC}(\theta)$$

Data Loss

The data loss is the mean square error between the NN output and the validation of known data points, it employs the data points to train the NN, so it's a supervised learning approach. The NN parameters are chosen by minimizing the difference between the observed outputs and the model's predictions for the observed inputs.

$$L_{data}(\theta) = \sum_{i=1}^{N} |u_\theta(x_i, t_i) - u_{exact}(x_i, t_i)|^2$$

Residual Loss

The residual loss is the mean square error between the NN output and the PDE residual. The PDE residual is the left-hand side of the PDE, which is computed using the automatic differentiation of the NN. It represents the loss produced by the mismatch with the governing physics laws defined by the PDE, it enforces the NN to satisfy the PDE at the collocation points, which can be chosen uniformly or unevenly over the domain

$$L_{physics}(\theta) = |PDE(u_\theta(x_i, t_i))|^2$$

Periodic Boundary Conditions and Initial Condition Loss

Periodic boundary conditions in the space domain are defined as it follows:

$$L_{SBC} = \sum_{i=1}^{N} |u_\theta(x_{min}, t_i) - u_\theta(x_{max}, t_i)|^2$$

Periodic boundary conditions in the time domain are defined as it follows:

$$L_{TBC} = \sum_{i=1}^{N} |u_\theta(x_i, t_{min}) - u_\theta(x_i, t_{max})|^2$$

And finally, the initial condition is defined as it follows:

$$L_{IC} = \sum_{i=1}^{N} |u_\theta(x_i, t_{min}) - u_0(x_i)|^2$$

Where u_0 is the initial condition function.

The loss function is minimized using the Adam optimizer, which is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing.

The physics constraints are included in the loss function to enforce model training, which can accurately reflect latent system nonlinearity even when training data points are scarce.

When solving PDEs using a numerical discretization technique, we are interested in the numerical method's stability, consistency, and convergence properties.

4.8.2 Parametric PINNs

PINNs can predict the variation in the solution for a range of parameters such as velocity, density, geometry, conductivity in our case the parameters are the mean μ and the variance σ for the Gaussian distribution and the amplitude A for the rectangular function, both of which are considered as the initial conditions of the PDE. By introducing them as features in the training data set, compared to conventional numerical solvers where each parameter needs a separate simulation and may require complex algorithms The idea is to add the parameter as another feature into the training data set such that each parameter has its own set of sampled points in the spatio-temporal domain.

4.8.3 Methodology

In our context t and x represent the time and space domains, we will train a neural network to approximate the solution of the PDE for all the discretization points in our domain at each time step by a multilayer feedforward neural network, we can do this by minimizing the error of the PDE in a certain number of points inside our domain.

We are going to represent our neural network by $\overline{u_\theta}$ then we define our initial condition as u_0 and our boundary condition as u_b then we define our $u\theta$ as follows:

$$u\theta(x,t,\beta) = u_0(x) + b_c(x)u_\theta^{NN}(x,t)$$

We assume then that:

$$\overline{u_\beta}(x,t,\theta) \approx u(x,t)$$

Where β is the set of parameters we want to optimize in order to minimize the error of the solution given by our neural network. It may seem strange to define the partial derivative of a neural network, but since the neural network's activation function is smooth and differentiable in our case we will be u using the **Tanh** activation function, we can derive it using automatic differentiation for any values of x and t.

We look to minimize θ in order to minimize the error of our neural network, we can do so by using a stochastic gradient descent-type (SGD) algorithm to optimize the parameter θ just as the standard training of deep neural networks. In our numerical examples, we use the Adam optimizer.

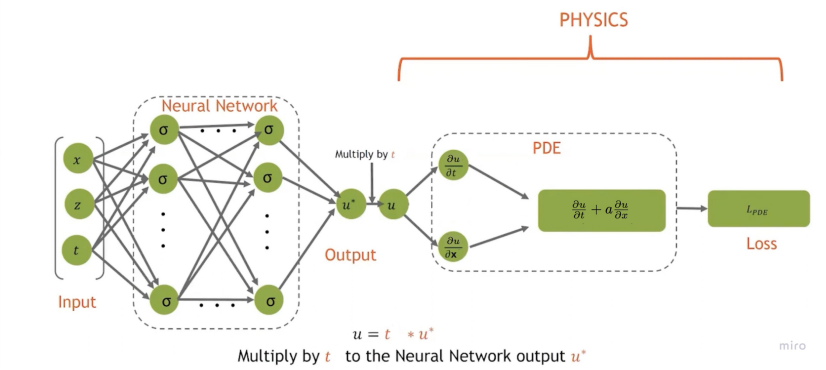


Figure 7: PINNs — **Source:** Youtube video: Introduction to PINNs

4.8.4 Convergence analysis

One of the goals for the implementation of the PINN theory is to investigate the convergence and stability of the computed u_θ to the exact solution of the

problem. They are related to how well the NN learns from physical laws and data.

5 Implementation

Our main objective is to implement a solver that based on the Semi-Lagrangian scheme, and implementing the Deep Lagrange interpolation operator using the u_θ function that will be approximated by a neural network that implements the PINNs strategy will solve the equation (1) for different initial conditions depending on the values of μ and σ for the Gaussian distribution and A for the rectangular function.

5.0.1 Neural Network Class

The Net class represents a neural network model. It inherits from nn.DataParallel, which is a PyTorch module used for parallelizing the computation on multiple GPUs. Our neural network contains 6 fully connected layers, an input layer that expects four or three input features x, t, mean, and variance/ A according to the initial condition, and five hidden layers, each consisting of fully connected linear units (nn.Linear) followed by the hyperbolic tangent activation function (torch.tanh). The hidden layers progressively reduce the input dimensions and extract higher-level features from the input data, and finally the output layer that consists of a single fully connected linear unit (nn.Linear) without an activation function. It maps the output of hidden layer5 to a single output value.

The forward method

Defines the forward pass of the neural network, where the input tensors propagate through the layers sequentially applying the specified operations and activation functions. The resulting output is returned as the final output of the network.

The networkBC method

Serves to impose the hard boundary condition, so when $t = 0$ we get that the solution at that time is equal to the initial solution.

5.0.2 Parameters class

This class defines the set of parameters for the PINN class including the initial condition u_0 defined as a Gaussian distribution or a rectangular function.

5.0.3 Network class

The Network class represents a **PINN** model.

1. **__init__(param: Parameters)**: Initializes the neural network model and loads the model if available.
2. **__call__(*args)**: Calls the network and returns the output.
3. **create_network()**: Creates the neural network model from the previously defined Net class.
4. **load(file_name)**: Loads the model from a file.
5. **save(file_name, epoch, net_state, optimizer_state, loss, loss_history)**: Saves the model with the specific values passed as arguments into a file.
6. **pde(x, t, mean, variance)**: Computes the PDE using the network and returns the result.
7. **predict_u_from_torch(x, t, mean, variance)**: Predicts the value of the solution given by the neural network based on the input variables x, t, `mean`, and `variance` or A
8. **random(min_value, max_value, shape, requires_grad=False, device=device)**: Generates random numbers within a range.
9. **make_data(n_data)**: Generates the data of size `n_data` for the training process based on the exact solution of the PDE.
10. **make_collocation(n_collocation)**: Generates `n_collocation` collocation points to enforce PDE constraints during training.
11. **train(epochs, n_collocation, n_data)**: Trains the neural network using a combination of PDE constraints and data fitting.
12. **u_exact(x, t, a, xmax, u0, mean, variance, device=device)**: Computes the exact solution for the PDE.
13. **plot(t, mean, variance)**: Plots the loss history, predicted solution, and prediction error at the input variables x, t, `mean`, and `variance`.

5.0.4 Optimization step

The **train** function implements an optimization method for training a neural network by combining PDE constraints and data fitting. It iterates over a number of epochs to update the network's weights and minimize the loss.

If there are collocation points specified, PDE constraints are enforced, collocation points are generated, and the network's output for these points is computed. Then the MSE loss is calculated between the network's output and a tensor of zeros, and this loss is added to the overall loss.

Similarly, if there are training data points specified, data fitting is performed, the raining data is generated, and then using the NN's prediction of the solution based on the input variables, then the MSE loss is computed between the

predicted solution and the exact solution, and this loss is added to the overall loss.

Additional loss terms are incorporated to enforce PDE constraints and boundary conditions. The network's solution is evaluated at the boundary points and compared to the boundary values to enforce periodicity in both spatial and temporal dimensions, then the loss is added to the overall loss.

Backpropagation is then performed to compute the gradients, and the weights of the neural network are updated using the gradient descent algorithm.

Throughout the training process, the current loss is recorded in the loss history and the model and optimizer states, as well as the loss history, are saved for further analysis. Upon completion of all epochs, the best model is saved.

5.1 Semi-Lagrangian solver

We define a class called **SemiLagrangianSolver** that contains the following methods:

1. u_0: To compute the initial condition.
2. **explicit solution**: To compute the analytical solution
3. **find closest**: To find the $n_{closests}$ points to x_*
4. **li**: To compute the Lagrange interpolation basis polynomial at x_*
5. **solver**: To calculate the numerical solution for the problem using the Lagrange interpolation operator.
6. **plot solution**: To plot the solution at a specific time
7. **error solution**: To calculate the error of the numerical solution compared to the exact solution
8. **u theta** The u_Θ solution found by the neural network.
9. **solver deep** to calculate the solution of the transport equation using the Deep Lagrange Interpolation and u_Θ.
10. **plot solution deep** to plot the solution deep at a specific time
11. **error solution deep** to calculate the error of the numerical deep solution compared to the exact solution

And the following parameters:

- nx the number of points in the space
- nt the number of points in the time
- a the velocity of the transport

- Δt the time step
- Δx the space step
- t_{min} the minimum time
- t_{max} the maximum time
- x_{min} the minimum space
- x_{max} the maximum space
- u_0 the initial condition
- u the solution

6 Results

Here we present the results of the PINNs implementation, we will show the results of the unsupervised and supervised learning, we will also show the results of the deep interpolation method.

6.1 Finding u_θ using PINNs

For training the neural network and solving the transport equation we used the following parameters:

- $min = 0.$
- $xmax = 1.$
- $tmin = 0.$
- $tmax = tf$
- $a = 1.$
- $learning\ rate = 1e-3$
- $min\ mean = 0.45$
- $max\ mean = 0.55$
- $min\ variance = 0.01$
- $max\ variance = 0.05$

This means that we will be learning the solution of the transport equation in the domain $[0, 1] \times [0, 1]$ with a velocity of $a = 1$ for all the Gaussian initial conditions that have a mean between $[0.45, 0.55]$ and a variance between $[0.01, 0.05]$.

6.1.1 Unsupervised learning

For the unsupervised approach we trained our NN with 40.000 epochs and 50.000 collocation points, the best loss we obtained was: **3.21e-04**

With a Gaussian initial condition with mean $\mu = 0.49$ and variance $\sigma^2 = 0.4$ we get the following approximation made by the network at different time steps:

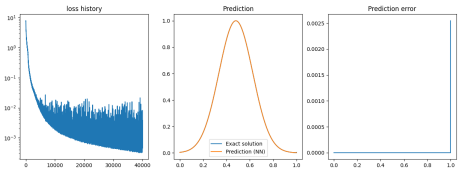


Figure 8: t=0

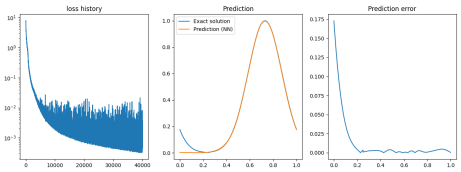


Figure 9: t=0.25

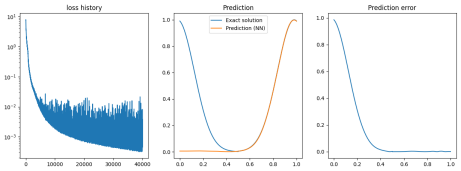


Figure 10: t=0.5

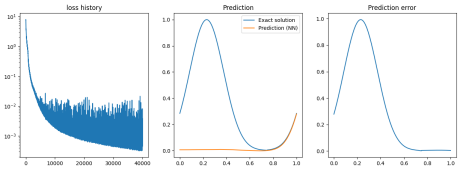


Figure 11: t=0.75

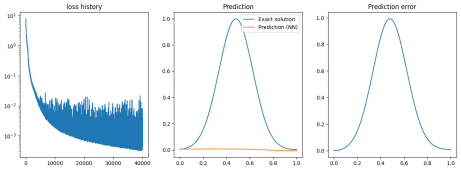


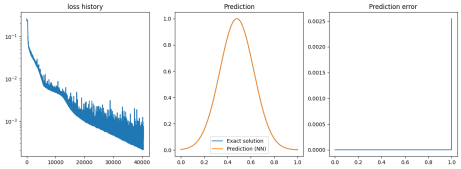
Figure 12: t=1

We can clearly see that by training our NN with collocation points it does not learn quite well all the detail of our solution. Even if the loss is small like in this case when implementing a PINNs strategy a small loss does not neccesarily equal to a good approximation.

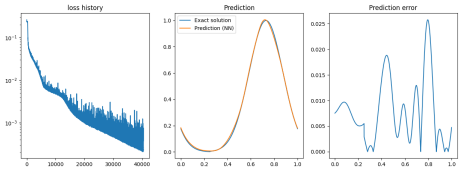
6.1.2 Supervised learning

For the supervised approach we trained our NN with 40.000 epochs and 15.000 data points, the best loss we obtained was: **3.76e-04**

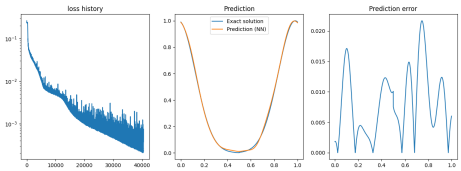
With a Gaussian initial condition with mean $\mu = 0.48$ and variance $\sigma^2 = 0.042$ we get the following approximation made by the network at different time steps:



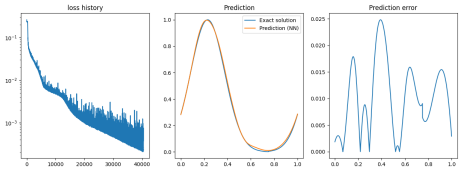
(a) t=0



(b) t=0.25



(c) t=0.5



(d) t=0.75

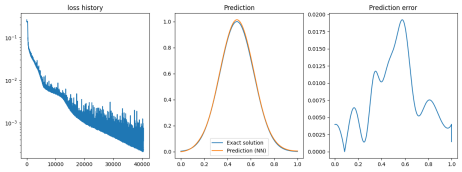


Figure 14: t=1

Error analysis

For 15.000 data points and different number of epochs the error of the NN is shown in the following table:

Epochs	Error
1000	7.28e-01
5000	2.86e-02
10000	4.12e-03
20000	1.95e-03
30000	1.25e-03
40000	3.76e-04

Table 1: Error of the NN prediction in for different number of epochs

6.1.3 Supervised and unsupervised learning

In the following figures we show the results of the PINNs implementation, we will show the results of the unsupervised and supervised learning.

The training was done using the following parameters:

- epocs = 40000
- ncollocation = 50000
- ndata = 10000

The best loss we obtained was: **7.80e-04**
For an initial solution with mean=0.5 and variance = 0.035 we get the following approximation made by the network at different time steps:

$$u_0(x) = \exp^{\frac{-(x-0.5)^2}{0.035}}$$

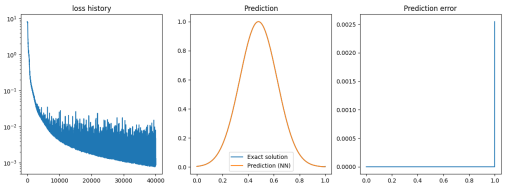


Figure 15: t=0

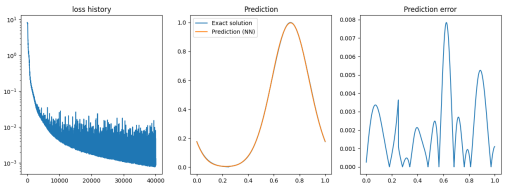


Figure 16: t=0.25

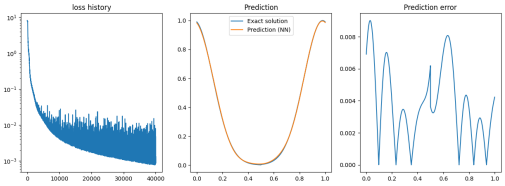


Figure 17: t=0.5

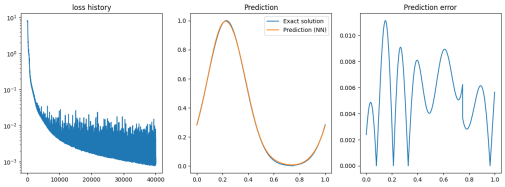


Figure 18: t=0.75

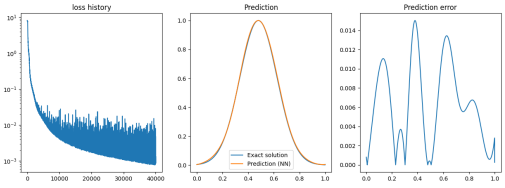


Figure 19: t=1

Error analysis

For 15.000 data points and different number of epochs the error of the NN is shown in the following table:

Epochs	Error
1000	3.30e-01
5000	1.36e-02
10000	5.11e-03
20000	1.95e-03
30000	8.35e-04
40000	7.80e-04

Table 2: Error of the NN prediction in for different number of epochs

6.2 Semi-Lagrangian solver using deep interpolation

The following results were obtained using the following parameters:

- $xmin = 0.0$
- $xmax = 1.0$
- $tmin = 0.0$
- $tf = 1.0$
- $nt = 100$
- $a = 1$
- $mean = 0.48$
- $variance = 0.042$

We obtained the following errors calculated in L2 norm compared to the exact solution (in loglog scale) of the transport equation according to the initial condition defined by the Gaussian with the mean and the variance given in parameter at times tf/4 tf /2 and tf (with tf the final time)

The **SL classic** solution is obtained with the Semi-Lagrangian scheme and the classic Lagrange interpolation, **SL Deep** is calculated using this time the deep interpolation operator and u_θ as the solution given by the neural network, **Network** is the solution obtained by the network.

6.2.1 Interpolation of order 1

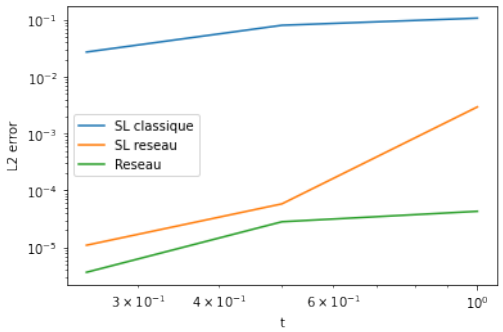


Figure 20: nx = 10

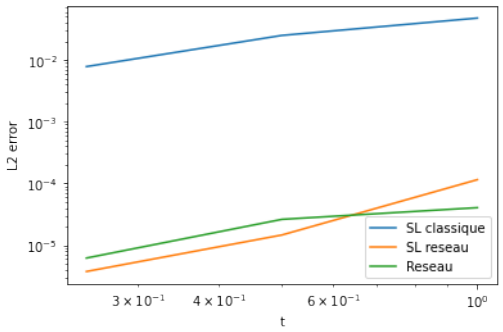


Figure 21: nx = 20

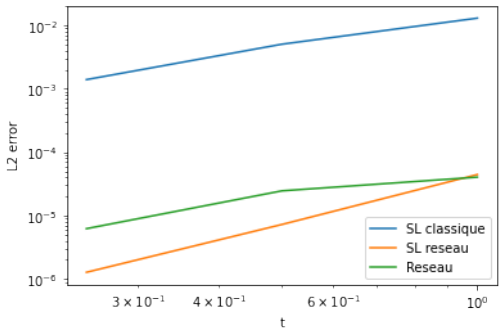
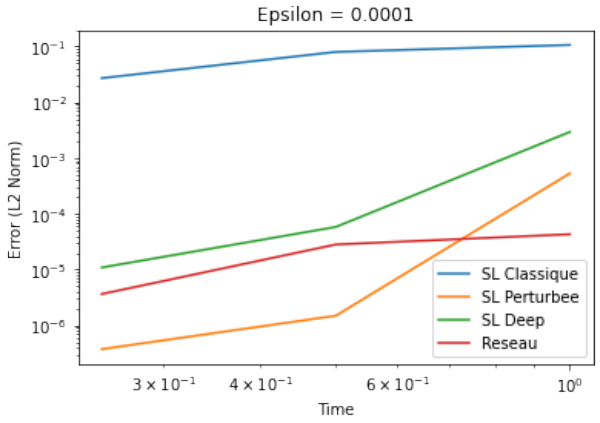
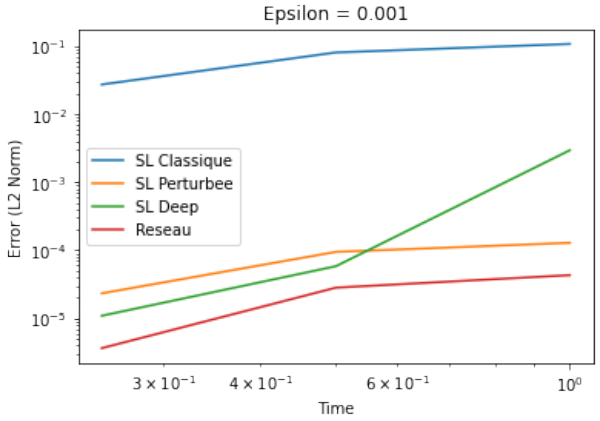


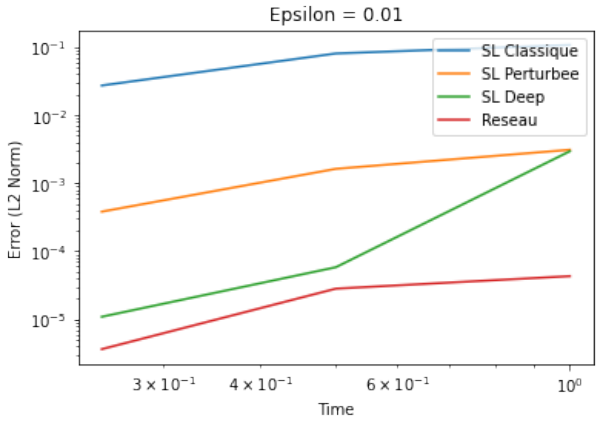
Figure 22: nx = 40

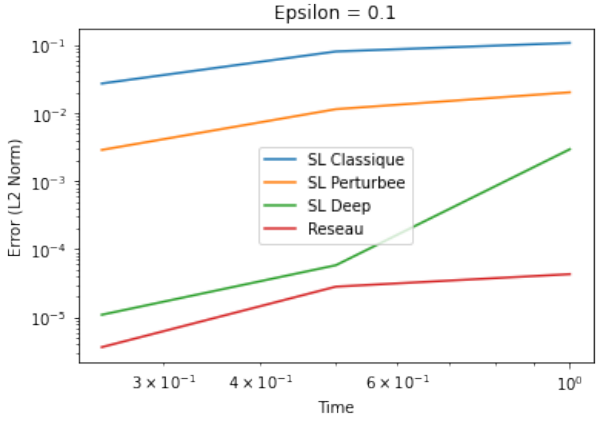
The solution **perturbed SL** is defined as: $u_{exact}(x,t)+\epsilon\cos(x)$ here is what we get when we vary the value of ϵ:

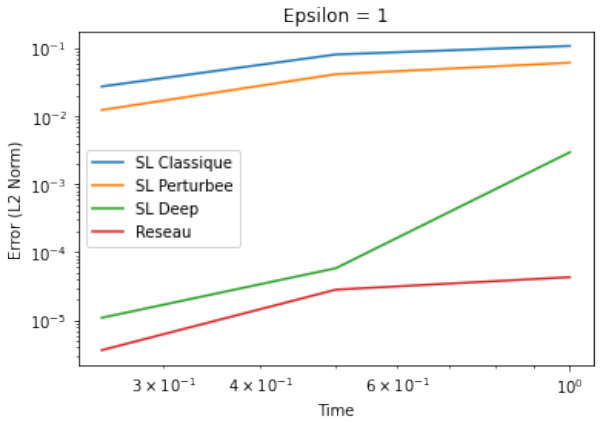
For nx=10


Epsilon = 0.0001
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

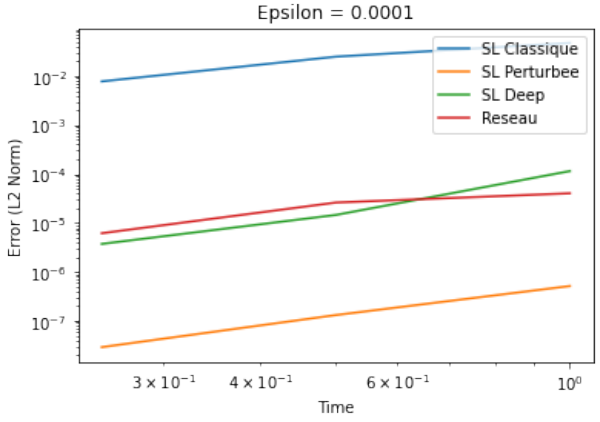

Epsilon = 0.001
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

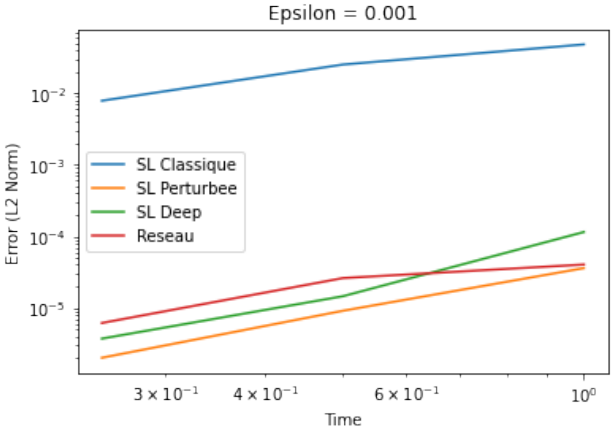

Epsilon = 0.01
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

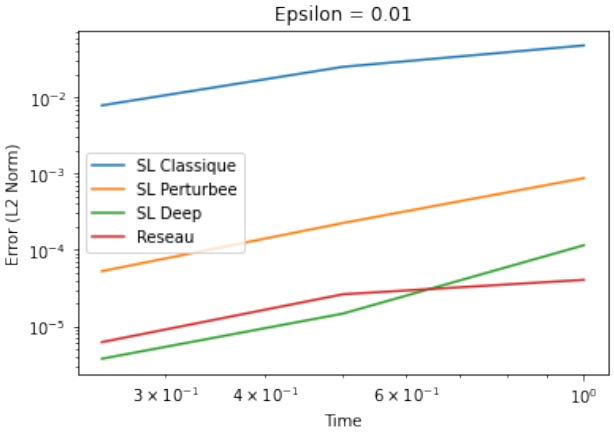

Epsilon = 0.1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

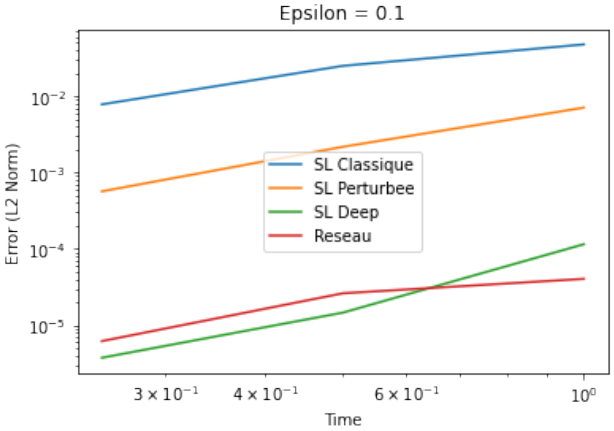

Epsilon = 1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

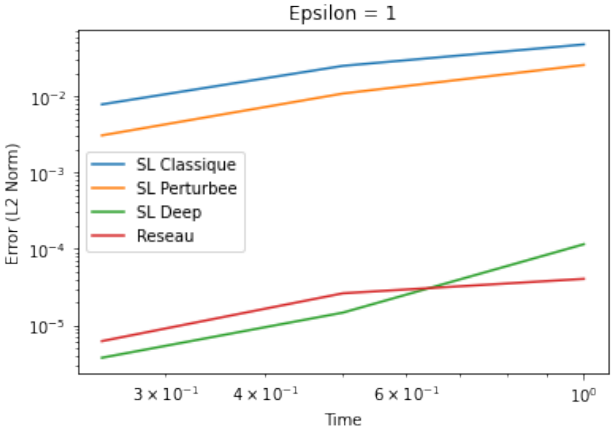
For nx=20


Epsilon = 0.0001
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

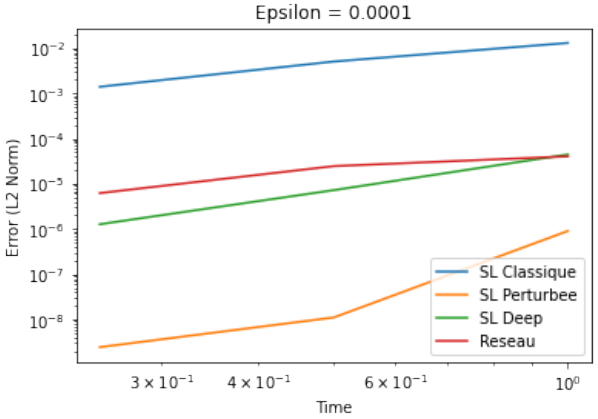

Epsilon = 0.001
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

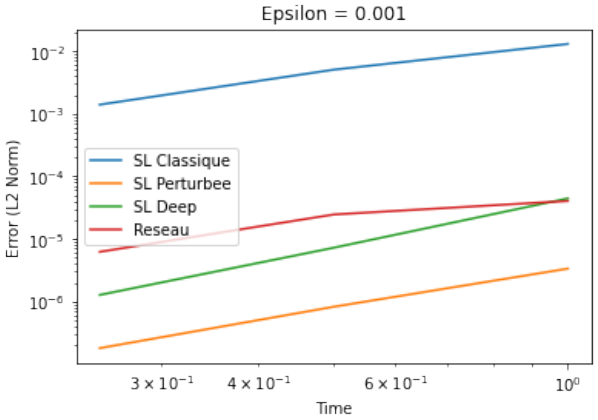

Epsilon = 0.01
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

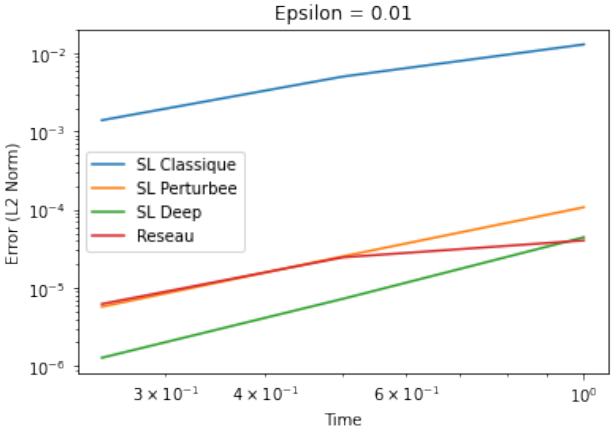

Epsilon = 0.1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

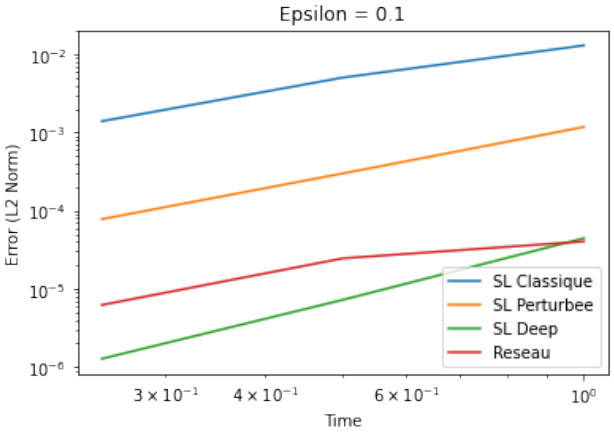

Epsilon = 1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

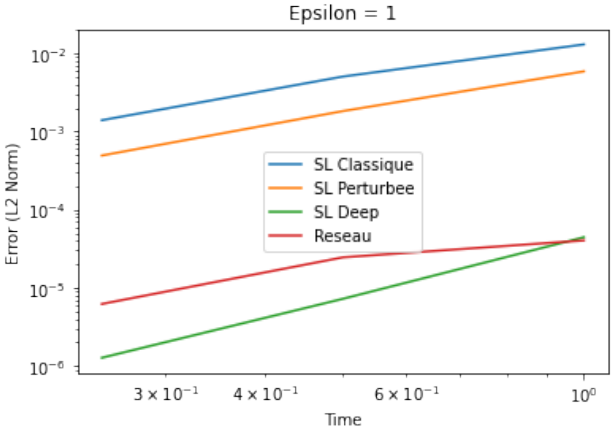
For nx=40


Epsilon = 0.0001
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau


Epsilon = 0.001
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau


Epsilon = 0.01
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau


Epsilon = 0.1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau


Epsilon = 1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

6.2.2 Interpolation of order 3

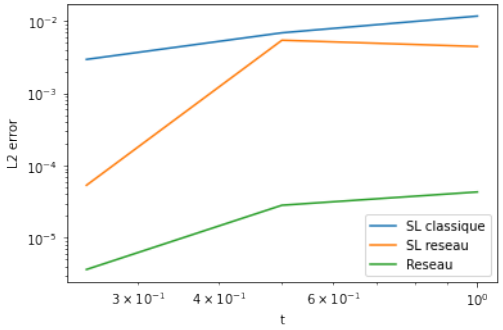


Figure 23: $nx = 10$

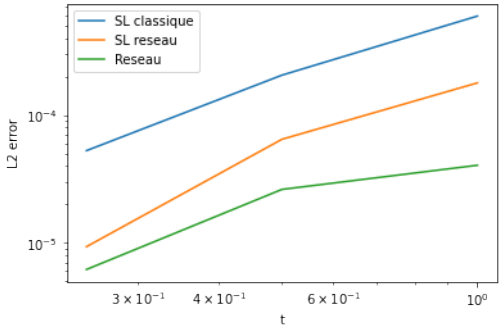


Figure 24: $nx = 20$

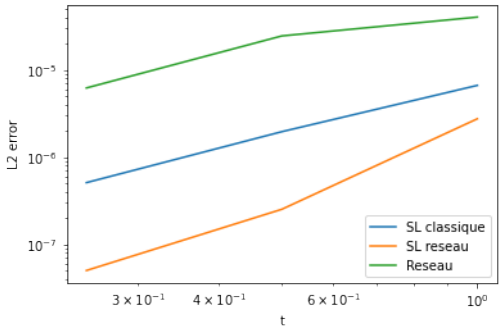
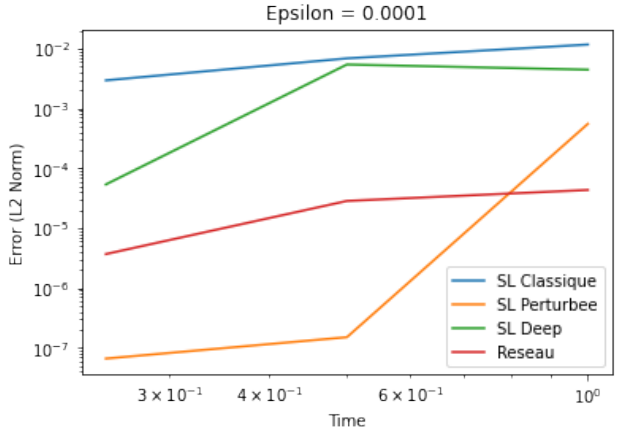
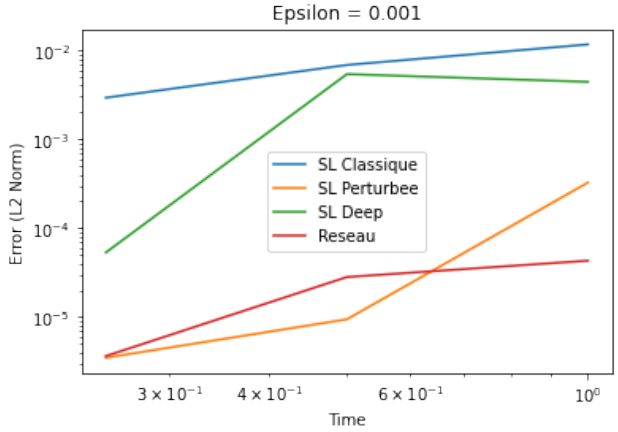


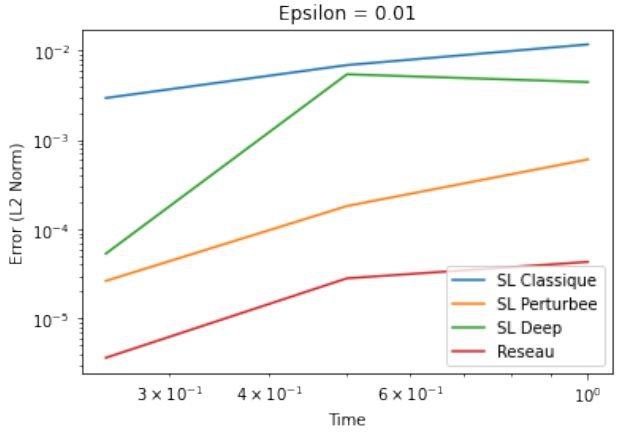
Figure 25: $nx = 40$

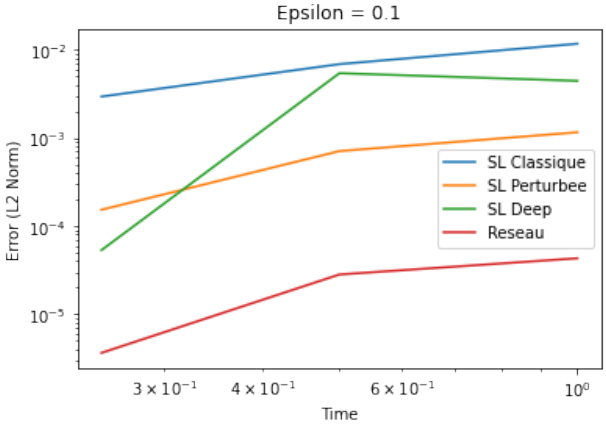
For the **perturbed SL** we get the following results:

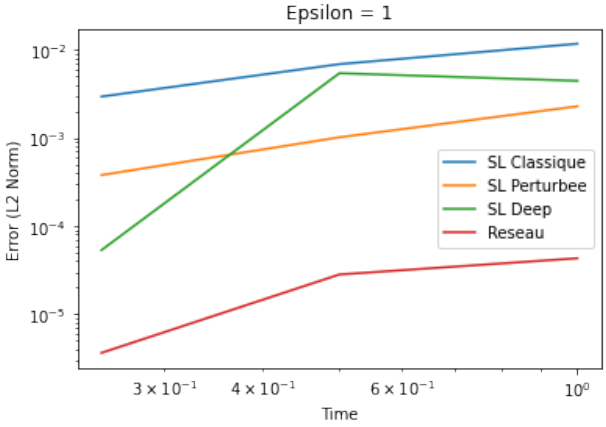
For nx=10


Epsilon = 0.0001
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

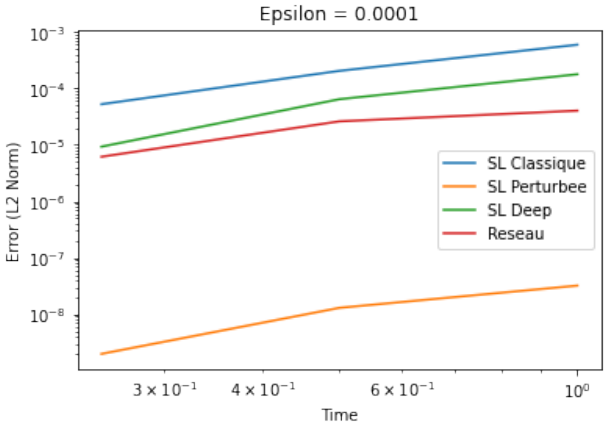

Epsilon = 0.001
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

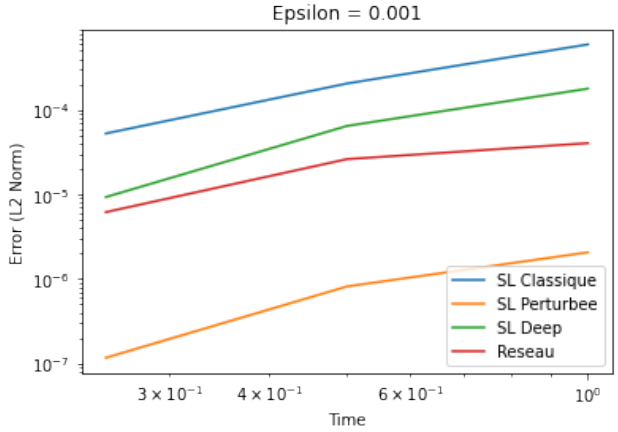

Epsilon = 0.01
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

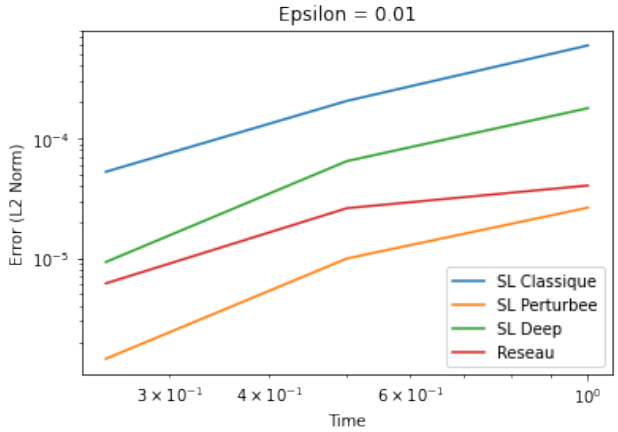

Epsilon = 0.1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

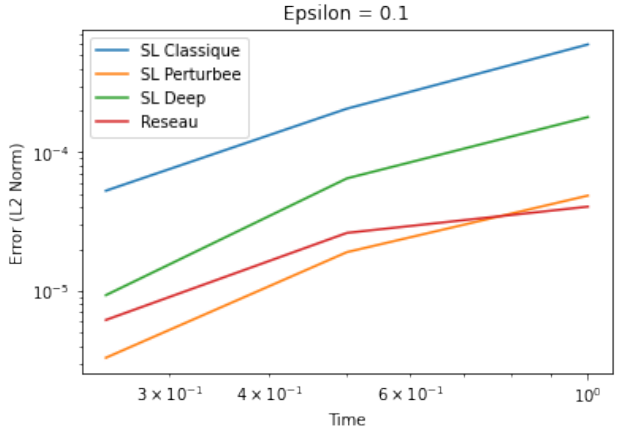

Epsilon = 1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

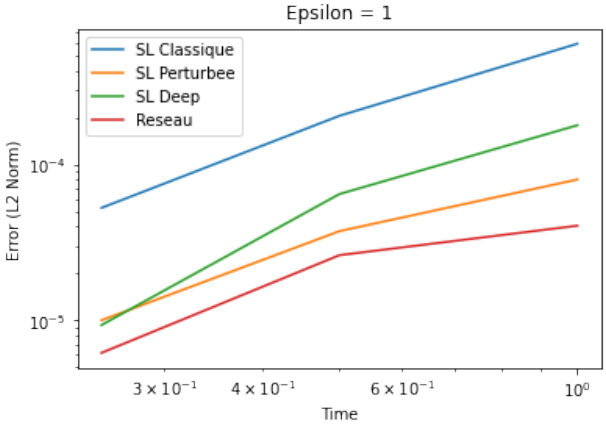
For nx=20


Epsilon = 0.0001
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

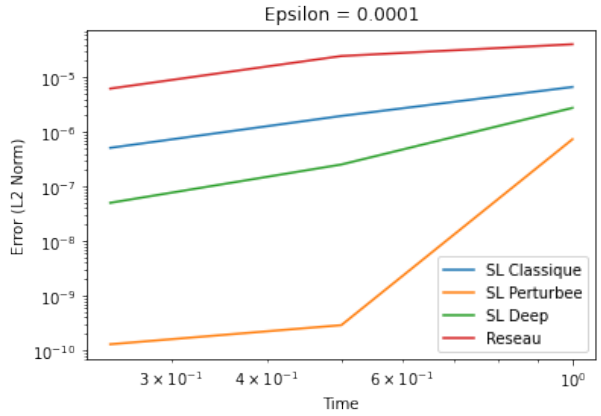

Epsilon = 0.001
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

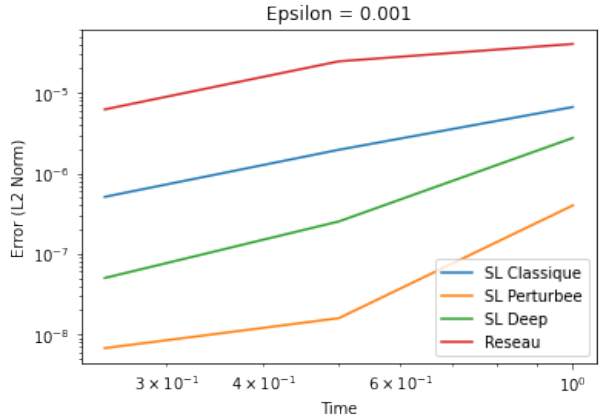

Epsilon = 0.01
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

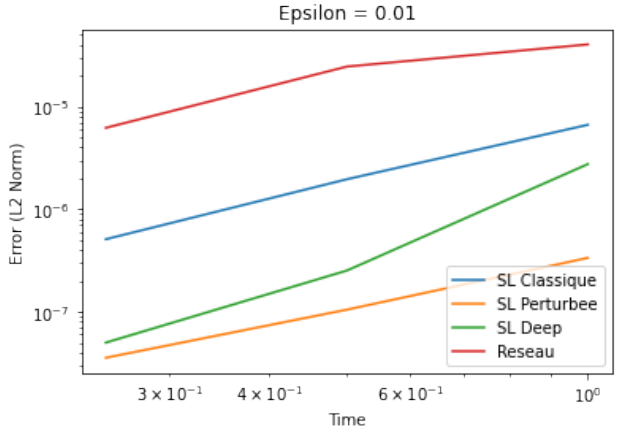

Epsilon = 0.1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

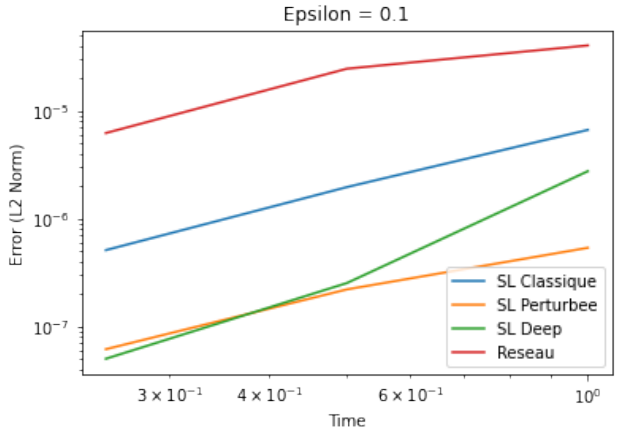

Epsilon = 1
SL Classique
SL Perturbee
SL Deep
Reseau
Error (L2 Norm)
10^{-4}
10^{-5}
3×10^{-1}
4×10^{-1}
6×10^{-1}
10^{0}
Time

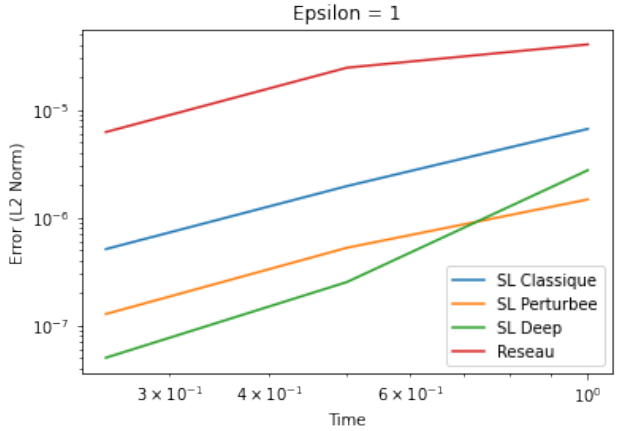
For nx=40


Epsilon = 0.0001
Error (L2 Norm)
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-10}
SL Classique
SL Perturbee
SL Deep
Reseau
3×10^{-1}
4×10^{-1}
6×10^{-1}
10^{0}
Time


Epsilon = 0.001
Error (L2 Norm)
10^{-5}
10^{-6}
10^{-7}
10^{-8}
SL Classique
SL Perturbee
SL Deep
Reseau
3×10^{-1}
4×10^{-1}
6×10^{-1}
10^{0}
Time


Epsilon = 0.01
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau


Epsilon = 0.1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau


Epsilon = 1
Error (L2 Norm)
Time
SL Classique
SL Perturbee
SL Deep
Reseau

6.3 Average gain

By randomly drawing 20 parameters for the mean and the variance to have different initial conditions, we calculate for each the solution with the classic and deep Semi-Lagrangian method and we look at the gain=SL error/deep SL error by doing the average over the 20 parameters.

By making several draws of 20 parameters, the following average gains are obtained:

For a 1st order interpolation: 137.7119541688143, 97.15674818809492, 185.7843283662372

For an interpolation of order 3: 7.608668520339034, 3.1395029764283313, 4.496461973620286

The gain is considerable when using a first order interpolation but much less important when using the classical third order interpolation.

7 Conclusions

By implementing the deep Lagrange interpolation operator alongside the u_θ solution derived through the PINNs strategy in the Semi-Lagrangian scheme for both the first and third interpolation degrees, the results demonstrate a notable reduction in the error as compared to the classical Semi-Lagrangian approach. Furthermore, we can see that the degree of improvement is considerably more pronounced when using the deep interpolation operator in with a first-order interpolation, as opposed to the utilization of the third-order one.
Additionally, our observations indicate that the PINNs strategy exhibit a remarkable capacity to assimilate a diverse range of solutions for the advection equation, accommodating varying initial conditions and parameters. This characteristic contributes to a highly accurate approximation of the solution.

References

[1] Cuomo, S., Di Cola, V. S., Giampaolo, F., Rozza, G., Raissi, M., & Piccialli, F. (2022). Scientific Machine Learning Through Physics–Informed Neural Networks: Where we are and What's Next.

[2] IBM Cloud Blog. (n.d.). Supervised vs. Unsupervised Learning: What's the Difference? Retrieved from https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning

[3] IBM. (n.d.). What are neural networks? Retrieved from https://www.ibm.com/topics/neural-networks

[4] DeepAI. (n.d.). Feed Forward Neural Network. Retrieved from https://deepai.org/machine-learning-glossary-and-terms/feed-forward-neural-network

[5] IBM Cloud. (n.d.). What is a feedforward neural network? Retrieved from https://www.ibm.com/cloud/learn/feedforward-neural-network

[6] Towards Data Science. (n.d.). Multilayer Perceptron Explained with a Real-Life Example and Python Code: Sentiment Analysis. Retrieved from: https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141

[7] *Proceedings of the National Academy of Sciences* (PNAS). (n.d.). Retrieved from https://www.pnas.org/doi/full/10.1073/pnas.1718942115

[8] Vadyala, S. R., Betgeri, S. N., Betgeri, N. P. (Ph.D). Advection equation using PyTorch. Department of Computational Analysis and Modeling, Louisiana Tech University, Ruston, LA, United States.

[9] Raissi, M., Perdikaris, P., & Karniadakis. Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations.