Thermal Storage Design with PINNs

Lorenzo Tarricone

April 12, 2024

1 PINNs for solving PDEs

For this task, I used the code provided in the tutorial as a "skeleton structure" onto which I developed my solution, as described in the handout. Therefore there is also here a `Pinn` class with the different methods to: sample, add and load into a dataloader the boundary points; apply the boundary condition using the NN approximation; compute the different loss and put them all together; train the model and plot the result.

1.1 NN implementation

I implemented the functional approximation of the two functions T_f and T_s as a single neural network that given a spatiotemporal input datapoint (x, t) would give as output the tuple of functional values $(T_f(x,t), T_s(x,t))$. In the final version of the code, the Neural network has 5 hidden layers and 20 neurons per layer.

1.2 Adding interior and boundary points

While the interior and the temporal boundary points are added similarly to the Colab tutorial, I decided to divide the spatial boundary points into single smaller "sub-datasets" that were then loaded independently as Dataloaders. In particular, I created two functions for the two kinds of boundary conditions used (namely, Dirichlet and Von Neumann) and for the latter, I returned different objects for the BC on the left of the domain and on the right of the domain.

1.3 Apply initial and boundary conditions

Once again I decided to adopt an approach "divide et impera" for my solution, meaning that I've written four different functions named (not surprisingly): `apply_initial_condition`, `apply_boundary_condition_D` (for Dirichlet), `apply_boundary_condition_VN_0` and `apply_boundary_condition_VN_L`

1.4 Compute PDE residuals and Calculate loss

This section was implemented in a similar way to the Colab tutorial. In the final version of the ode I've used a value of $\lambda = 100$

1.5 Model Training

The training was done for an instance of the `Pinn` class with 256 interior points and 64 spatial and temporal boundaries. I used ADAM as an optimizer with step size 0.001. The original training was done through 10000 epochs, but the resulting loss curve showed that probably was too much. I, therefore, decided to do early stopping and train for just 5000 epochs instead. This gave rise to the second loss curve

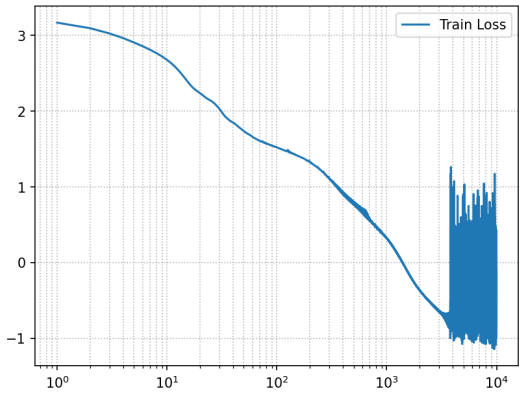

Figure 1: Training with 10k epochs (before)

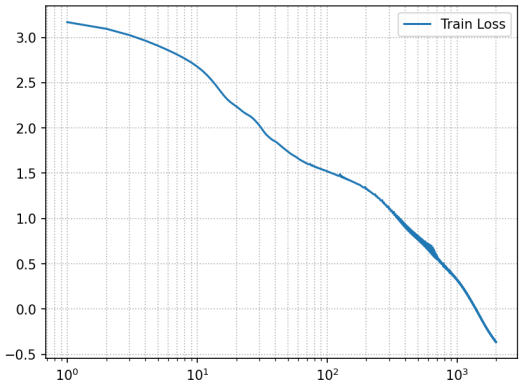

Figure 2: Training with 5k epochs (after)

1.6 Results

As a quality check, I've plotted the two approximate solutions for some points in the domain spatiotemporal domain $[0, 1] \times [0, 1]$. The results look promising as there seem to be no weird behaviours and the solution seems to agree with both the spatial and the temporal boundary conditions

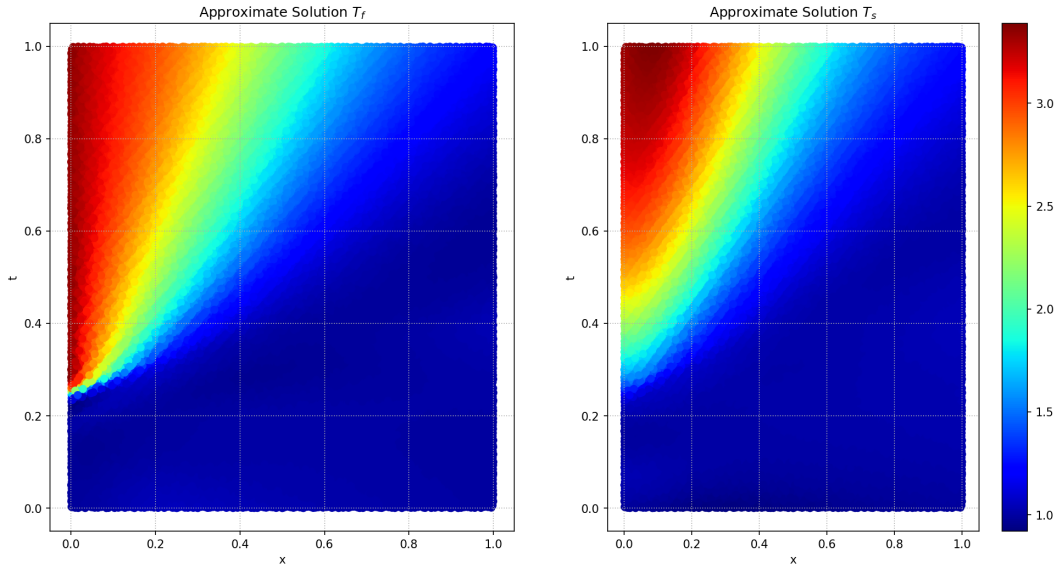

Figure 3: Functional approximations for task 1

2 PDE-Constrained Inverse Problem

For this second task, I also used the Colab code as a template. the name of the created class is `Pinn_inverse_problem_2)` (because I ended up adopting the second approach tried, see the following section).

2.1 NN implementation

I implemented the functional approximation of the two functions T_f and T_s as a single neural network that given a spatiotemporal input datapoint (x, t) would give as output the tuple of functional values $(T_f(x, t), T_s(x, t))$. In the final version of the code, the Neural network has 7 hidden layers and 42 neurons per layer. The original version of the code implemented two separate networks for T_f and T_s, but I've noticed that for me the training was faster and with better results when training just one network. My idea is that the single network solution is more able to learn the "complementary action" of the solid and the fluid in the thermal storage.

2.2 Adding interior and boundary points

The temporal boundary points are generated and given to the `DataLoader` altogether, while the spatial points are divided into the ones on the left-hand side of the domain and those on the right-hand side. To distinguish the different cycles on the spatial domain I extensively used the function `torch.where()` and set the correct target value for the spatial boundary at different positions. I moreover created a class function to visualize the points that will be used by the network to learn. In the final design, I used 2096 interior points and 512 spatial and temporal boundary points

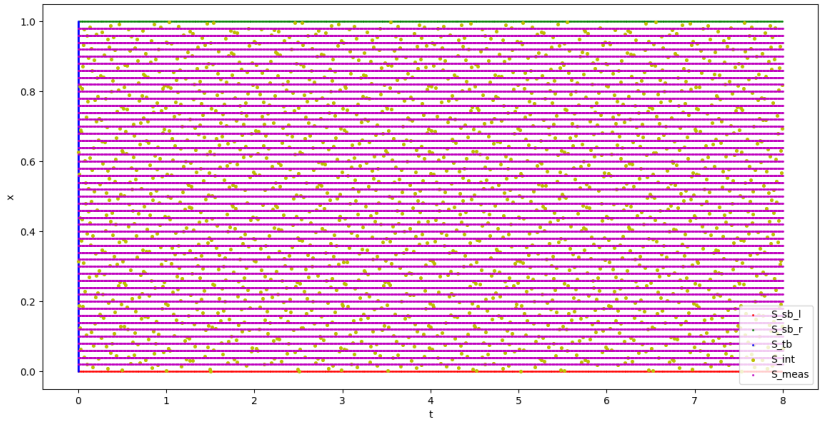

Figure 4: Points used during the training

2.3 Apply initial and boundary conditions

For the application of the initial and boundary conditions, I continued to use the function `torch.where()` to make sure to get the correct result (functional approximation value for T_f or its derivative).

2.4 Other functions

Two helper functions were implemented

- `Get_U_F` given the input values (of the interior points) looks at the time column and returns the correct vector of values in $\{0, 1, -1\}$ according to the corresponding phase of the process, as indicated in the handout
- `get_measurement_data` reads the data for the experimentally measured values of T_f and the corresponding points in the spatiotemporal lattice and returns them as `torch` tensors

2.5 Compute PDE residuals and Calculate loss

This section was implemented in a similar way to the Colab tutorial. In the final version of the ode I've used a value of $\lambda = 10$

2.6 Model Training

For the training, I used the LBFGS optimizer with a `max_iter` parameter of 10000. Apart from some spikes in the loss curve towards the end of the training the loss steadily decreased.

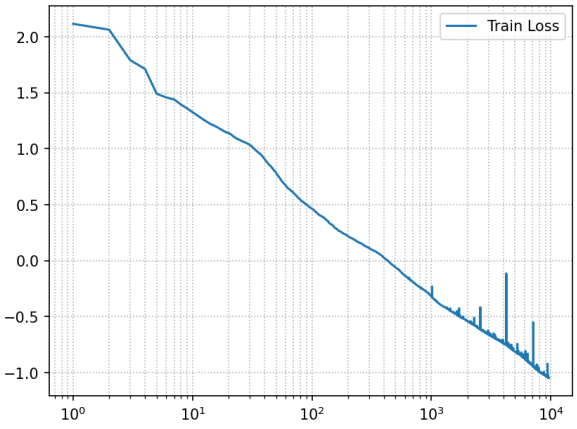

Figure 5: training loss for task 2

2.7 Results

As a simple quality check, I plotted the measured values of T_f next to the functional approximation learned by my Neural Network and calculated some statistics of the error. With the training process described above I obtained a a relative error of 3.304% with a mean error of 0.035 and a standard deviation of 0.049

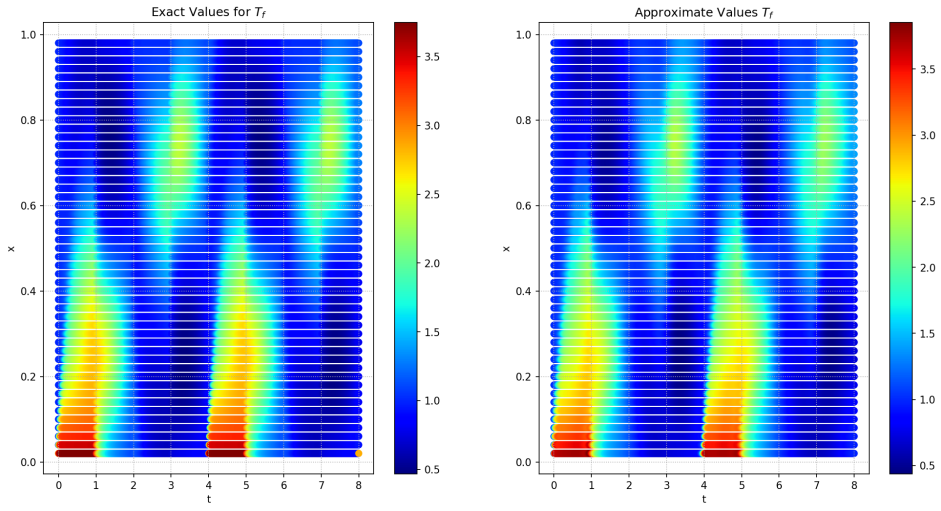

Figure 6: training loss for task 2

As a sanity check, I have also plotted the approximated function T_s (our final objective) next to the approximated T_f. We can see how the heat is transferred to the solid in the charging phase and how the transfer is reversed in the discharging phase. Apart from some small anomalies in the bottom of the lattice (around $(t, x) = (4, 0)$) the solution learned seems realistic

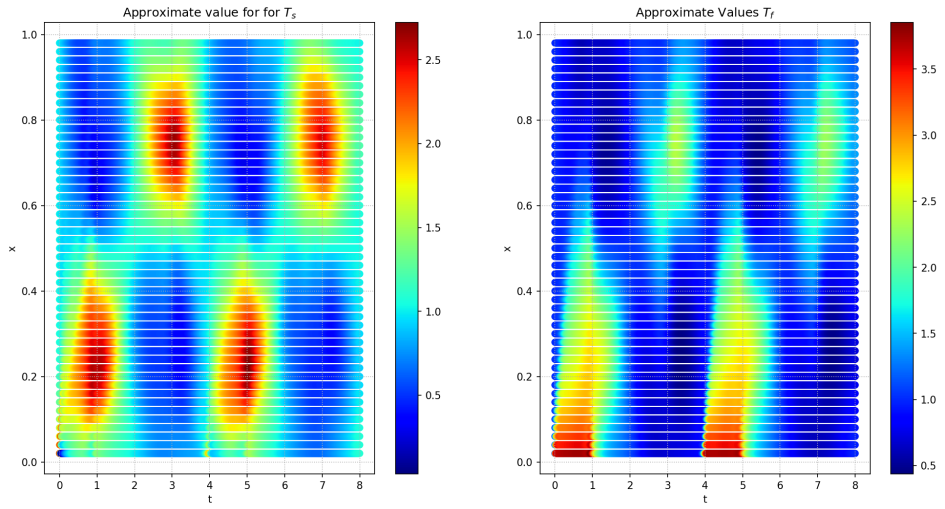

Figure 7: Comparison of the functional approximations of T_s and T_s learned by the NN

3 Applied Regression

3.1 Data Undestanding

To better understand the data I proceeded with some standards and procedures of data exploration, I also did some data curation intending to facilitate the training of the required Neural Network. The library `pandas` makes it very easy to explore these tabular features and manipulate these data.

With the `.info()` function we can already see elements of heterogeneity in the data.

- the feature `ocean_proximity` is not of type `float64` but are strings indicating one of these options: "<1h ocean", "inland", "near ocean", "near bay", "island"
- the feature `total_bedrooms` has 207 null values

Before facing these two problems, I plotted the histogram of the distribution of the numerical features together with the variable to predict: `median_house_value` (in the bottom right) 8

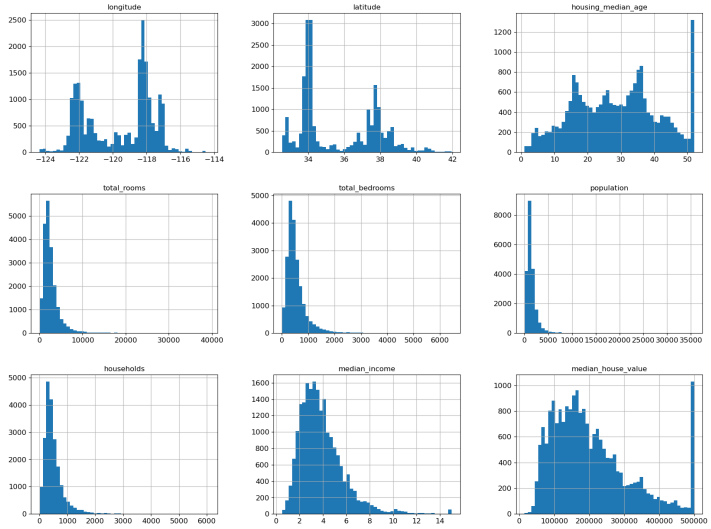

Figure 8: histograms of the numerical features + values to predict

We can see that many features have a quasi-gaussian distribution, possibly with some skewness on the right side. Notable exceptions are the features in the first row of 8. Geographical features present

both two peaks, indicating two possible regions that are densely populated in California (which is indeed the case looking at the simple plot 9

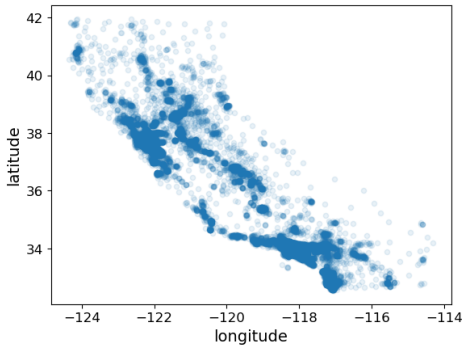

Figure 9: Plotted latitude and longitude

The shape of the histogram of the feature `housing_median_age` is probably caused by more complex factors like the demand for houses over time or the economic condition of the time. Another non Gaussian distribution is the target variable, which has an unproportionate number of values around 500000. I suppose this is due to the fact that the value over 500K was clipped onto this value.
Correlation plots also show that some features are (as expected) very correlated, such as `total_rooms` and `total_bedrooms`

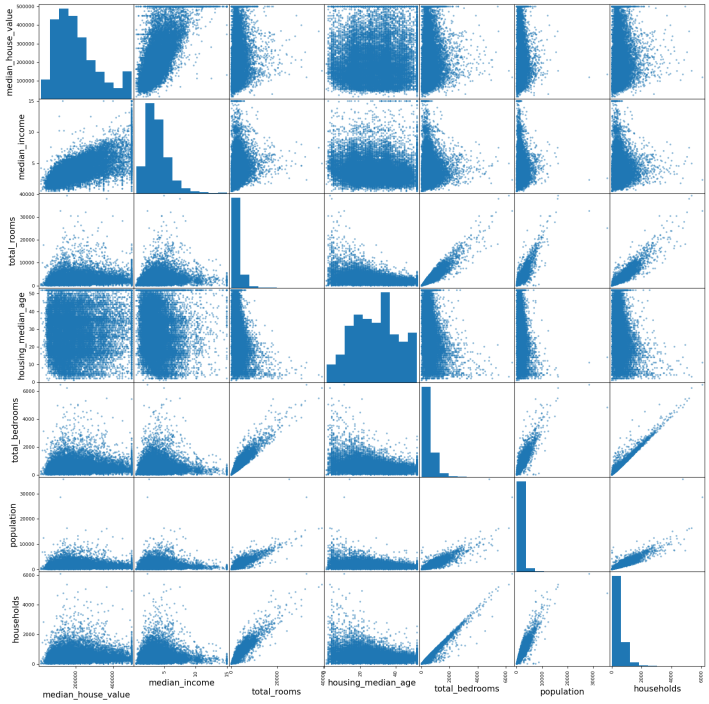

Figure 10: Correlation plot

Coming back to the two issues underlined above, I state that I can justify the elimination of the feature `ocean_proximity` because, in the rest of the data, there is enough information to have good predictions using a Neural Network. To give credit to this position, I performed a dimensionality reduction with `t_SNE` of the data points taking out the target variable. (for this step I've dropped all the data points that presented some N.A.) 11. It seems that the z coordinate in the plot is partially correlated with the target variable, therefore a neural network that is expressive enough would be able to catch this pattern.

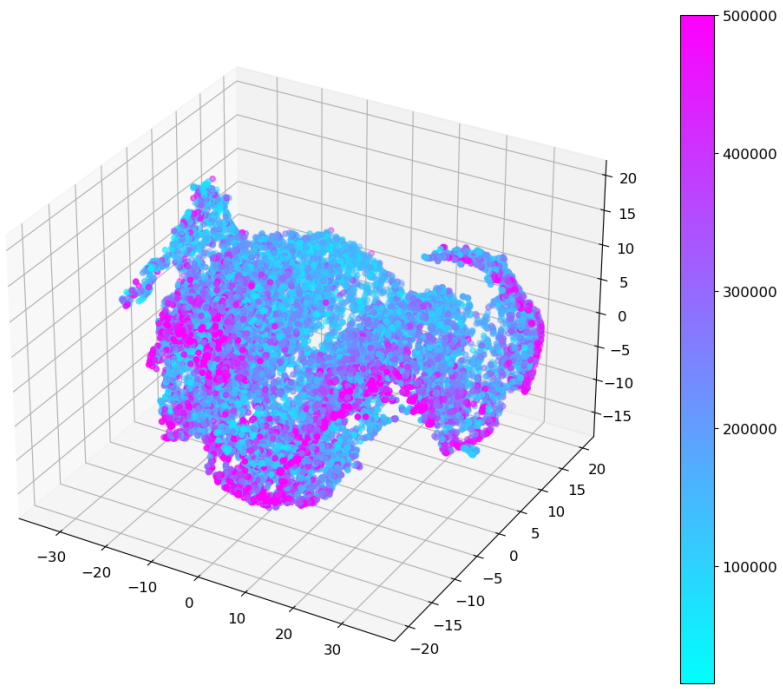

Figure 11: t-SNE plot of the features, coloured by the target variable

To solve the second problem I used linear regression of the features `total_rooms`, `households` and `population` to predict the missing values of `total_bedrooms`, because these two features seemed to be the three most correlated features with the latter form the pairwise correlation plot.

3.2 Model Implementation

To select the best neural network architecture and hyperparameters I performed a grid search with the following possibilities:

- number of hidden layers: $[1, 2, 3, 4]$
- number of neuron per layer: $[50, 100, 200, 300]$
- activation function: [relu, sigmoid, tanh]

All the models were trained with Optimizer Adam (learning rate: 0.001) and for 1000 epochs
The top three models according to the metric "RMSE on the test set" were (best first):

- $[2, 100, relu]$
- $[3, 50, relu]$
- $[2, 200, relu]$

while the worst three models were (best first)

- $[3, 50, sigmoid]$
- $[4, 50, sigmoid]$
- $[2, 50, sigmoid]$

This result shows that (at least for this amount of epochs and this optimizer) the ReLu function and a mid-sized network seem to provide the best results on the test set. On the other side, small networks trained with a sigmoid activation struggle to learn the patterns in the data.
After that, I trained a network with the best set of hyperparameters ($[2, 100, relu]$) for 10000 epochs 12 and this gave as RMSE on the test set a value of 56464.1523

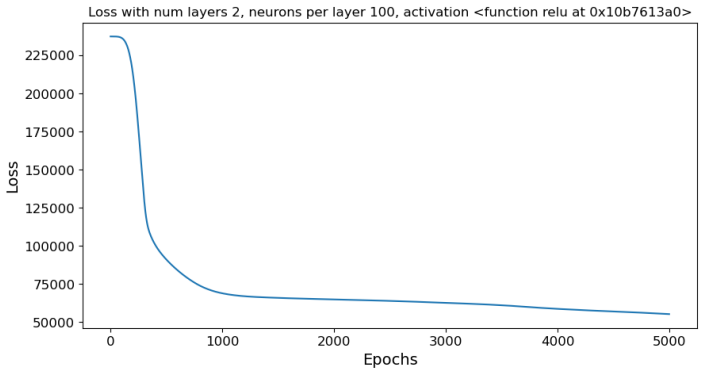

Figure 12: Loss curve for the new training of the best model

3.3 Comparison with reference

In the linked notebook after a detailed data description, analysis and processing, some Good Old Fashion Artificial Intelligence (GOFAI) methods were tried, namely:

- **Ridge regression** (with baseline RMSE of 68701.7198 later improved to 65418.0158 including some insights on the data and 63980.8275 with some non-linear combinations of features
- **Random Forests** (with RMSE baseline of 47825.1058, improved to 45594.7621 after grid search)
- **Gradient Boosting** (with RMSE baseline of 46429.3630, improved to 44255.9615 after complexity optimization with GridSearch)

My Neural Network performs a bit worse than the three-based methods, but much more could be done in order to bring its performance to comparable results. For example, some form of regularization will probably make the model generalize better as Deep NNs tend otherwise to overfit the training data.
In general, though, the performances will be comparable to those of the "white box models" Random Forest and Gradient Boosting, leading me to conclude that we can indeed predict the target variable with a neural network, but that in practice it might be an overkill and GOFAI methods can be as good in terms of performance and might give much more insights about important features (or combination of features). The linear regression (even with nonlinear features) is probably not expressive enough to catch the nonlinear patterns in the data. We could also have probably guessed it from the complex pattern displayed in the `t_SNE` plot

4 Robusness of PINNs and transferability

4.1 Bound on error

Given the the trained PINN u_θ we can write the following chain of equalities:

$$
\begin{aligned}
&||u_\theta - u_\delta||_{C^1(B_{1/2})} = ||u_\theta - u + u - u_\delta||_{C^1(B_{1/2})} <= \\
&||u_\theta - u||_{C^1(B_{1/2})} + ||u - u_\delta||_{C^1(B_{1/2})} <= ||u_\theta - u||_{C^1(B_1)} + ||u - u_\delta||_{C^1(B_{1/2})} <= \\
&\epsilon + C_\epsilon(||u - u_\delta||_{L^\infty(B_1)} + ||f - (f + \delta\phi)||_{L^\infty(B_1)})
\end{aligned}
$$

Where the first term of the last equation is given by hypothesis and the second is because $u - u_\delta$ satisfies $\nabla \cdot (A\nabla[u - u_\delta])$ so we get:

$$
\begin{aligned}
&\nabla \cdot (A\nabla[u - u_\delta]) = \\
&\nabla \cdot (A\nabla u - A\nabla u_\delta]) = \\
&\nabla \cdot (A\nabla u) - \nabla \cdot (A\nabla u_\delta) = f - (f + \delta\phi)
\end{aligned}
$$

Using the linearity of the differential and the divergence operators.
Last, applying the definition of L^∞ norm gives:

$$
\epsilon + C_\epsilon(\sup_{B1} u - \sup_{B1} u_\delta + \sup_{B1} \delta\phi)
$$

4.2 Robustness of PINNs solution and transfer learning

We have seen in lecture 6 how PINNs can be slow in solving forward problems and how a change in the source (in that case the position of the source of the wave) could make the prediction for the new source very bad. An idea presented to solve that was "conditional PINNs" where the source was given as additional input of the NN approximating the solution. This would mean that now the Network a representation of the solution that can generalize well for many sources. Looking on the web, I've found the SVD-PINNs architecture. The SVD-PINNs paper proposes a transfer learning strategy for PINNs by keeping singular vectors and optimizing singular values. This approach allows PINNs to solve a class of PDEs with different but closely related right-hand-side functions and proves the effectiveness of transfer learning in reducing the computational cost associated with training PINNs for new tasks.

Changing the initial data (for example, more noisy measurement data when using PINNs to solve an inverse problem like task 2) can also lead to problems in the generalization capabilities of the PINNs because you would require more samples to have a good estimate of the true underlying function (i.e. you would have a bigger estimation error [lecture 3])

4.3 Differential operator and robustness

The complexity of the differential operator is impacting the performances of the model as a more complex operator would imply that the DNN will have to learn a representation of more derivatives. It can be found in the literature the generalization error is directly linked to the number of derivatives in the differential operator

$$
\epsilon_G(\theta) <= C(u, u_\theta)||u - u_{\hat{\theta}}||_{W^{p,s}}
$$

Where s depends on the number of derivatives in the differential operator.
It was proven in 2021 by Mishra and Molinaro that Coercivity of the function u is a sufficient condition to have a bound of total error in terms of residuals ($||u - \hat{u}||_p <= C_{pde}(\hat{u}, u)||D(\hat{u}) - D(u)||_p$) so if I have control on the solution I can ensure this bound.

In general, it's useful to control the source term because it impacts the robustness of the training (see point above) and the number of initial datapoints because it will determine the quadrature approximation error.