PINO applied to 2D wave equations

Ugo PELISSIER

March 13, 2023

1 Problem description

The acoustic wave equation for the square slowness, defined as $m = \frac{1}{c^2}$ where c is the speed of sound (velocity) of a given physical medium with constant density, and a source q is given by:

$$u_{tt} = c^2(u_{xx} + u_{yy}) + q$$

Where $u(x,t)$ represents the pressure response (known as the "wave-field") at location vector x and time t in an acoustic medium. Despite its linearity, the wave equation is notoriously challenging to solve in complex media, because the dynamics of the wave-field at the interfaces of the media can be highly complex, with multiple types of waves with large range of amplitudes and frequencies interfering simultaneously.

2 Neural Operators

Neural operators use neural networks to learn operators rather than single physical systems. In our case, PDEs are the operator these networks try to learn. Specifically, we provide neural operators with input fields $\mathcal{A}$ that are composed of coordinates and relevant data such as coefficient fields, ICs, and BCs. Neural operators then output the solutions of that operator at those coordinates which we will denote as $\mathcal{U}$. We can mathematically represent the neural operator $\mathcal{G}_\theta$ as a mapping between the input fields $\mathcal{A}$ and the output fields $\mathcal{U}$ as

$$\mathcal{G}_\theta : \mathcal{A} \to \mathcal{U}$$

where θ are the weights of the neural operator.

PINOs are a variation of neural operators that incorporate knowledge of physical laws into their loss functions. PINO loss function is described by:

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \mathcal{L}_{\text{pde}}$$

3 Training data generation

To generate training data, we took the random IC fields that we had previously generated and evolved them in space and time. Specifically, we evolve each of these equations in time with RK4 time stepping starting at $t = 0$ until $t = 1$ with the time-step δt varying depending on the problem. To compute spatial derivatives, we employed a finite difference method (FDM) with 4th order central difference for most cases.

To match the Darcy case set-up in Modulus, 1000 training examples and 100 validation examples (fig. 1).

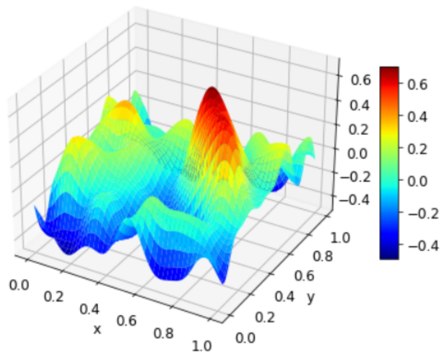


Figure 1: Training example

4 Case set-up

In our case, we want to learn a mapping between the initial conditions and the time-dependant solution:

$$ICs \rightarrow \mathcal{U}(x,t)$$

The PDE loss will be defined and the Neural Operator model will be contrained using it. This process is described in detail in Defining PDE Loss below.
Note that the PDE loss involves computing various partial derivatives. In general this is nontrivial; in Modulus, three different methods for computing these are provided. These are based on the original PINO paper:

- Numerical differentiation computed via Finite-Difference Method (FDM)
- Numerical differentiation computed via spectral derivative
- Hybrid differentiation based on a combination of first-order "exact"2 derivatives and second-order FDM derivatives

The first 2 approaches are the same as proposed in the original paper. The third approach is a modification of the "exact" approach proposed in the paper. This method is slower and more memory intensive than the numerical derivative approaches when computing second order derivatives because it requires the computation of a Hessian matrix. Instead, a "hybrid" approach is provided which offers a compromise by combining first-order "exact" derivatives and second-order FDM derivatives.

5 Defining PDE Loss

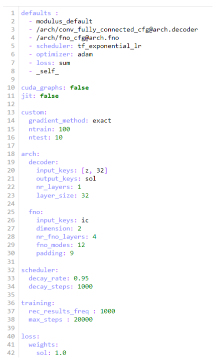
Using the Modulus example for the Darcy problem, the code has been adapted to fit the Wave Equation problem. However, it was not possible to add the time derivative to the framework provided by Modulus for PINO. In order to be able to run simulations, only the final time step was kept in the solution and the stationary problem was solved, assuming a time derivative equal to 0.

```
outvar_train['sol'] = np.array(outvar_train['sol'][:,100:,:,:])
outvar_test['sol'] = np.array(outvar_test['sol'][:,100:,:,:])
```

```
class Wave(torch.nn.Module):
    "Custom Wave PDE definition for PINO"

    def __init__(self, gradient_method):
        super().__init__()
        self.gradient_method = str(gradient_method)

    def forward(self, input_var: Dict[str, torch.Tensor]) -> Dict[str, torch.Tensor]:
        # get inputs
        u = input_var["sol"]
        ic = input_var["IC"]
        c = 1.0

        dxf = 1.0 / u.shape[-2]
        dyf = 1.0 / u.shape[-1]

        if self.gradient_method == "hybrid":
            dduddx_fdm = input_var["sol__x__x"]
            dduddy_fdm = input_var["sol__y__y"]
            # compute wave equation
            wave = (
                c**2 * (dduddx_fdm + dduddy_fdm)
            )
        else:
            raise ValueError(f"Derivative method {self.gradient_method} not supported.")

        # Zero outer boundary
        wave = F.pad(wave[:, :, 2:-2, 2:-2], [2, 2, 2, 2], "constant", 0)
        # Return darcy
        output_var = {
            "wave": dxf * wave,
        }  # weight boundary loss higher
        return output_var
```

6 Configuration

The configuration for this problem is fairly standard within Modulus. A specific FNO architecture is defined for this example inside of the configuration file (fig. 2). These settings were derived through an automated hyper-parameter sweep using Hydra multirun. The most important parameter for FNO models is *dimension* which tells Modulus to load a 1D, 2D or 3D FNO architecture. *nr_fno_layers* are the number of Fourier convolution layers in the model and *fno_layer_size* are the size of the latent embedded features inside the model.



```yaml
defaults :
  - modulus_default
  - /arch/conv_fully_connected_cfg@arch.decoder
  - /arch/fno_cfg@arch.fno
  - scheduler: tf_exponential_lr
  - optimizer: adam
  - loss: sum
  - _self_

cuda_graphs: false
jit: false

custom:
  gradient_method: exact
  ntrain: 100
  ntest: 10

arch:
  decoder:
    input_keys: [z, 32]
    output_keys: sol
    nr_layers: 1
    layer_size: 32

  fno:
    input_keys: ic
    dimension: 2
    nr_fno_layers: 4
    fno_modes: 12
    padding: 9

scheduler:
  decay_rate: 0.95
  decay_steps: 1000

training:
  rec_results_freq : 1000
  max_steps : 20000

loss:
  weights:
    sol: 1.0
```

Figure 2: Configuration file

7 Results

The TensorBoard plot (fig. 3) shows the relative error of the prediction compared to the exact solution. We observe a rapidly decaying error at the begining, reaching a plateau at the end of the training at 0.1. This error is still relatively high in comparison to the reference of the Darcy case, where the final error was about 0.01.

Thanks to TensorBoard, we can visualize at the same time the plots of the prediction field compared to the exact solution (fig. 4). We observe, from a qualitative point of view, that the prediction indeed reproduces the main patterns of the exact solution. However, when looking in detail, we observe that the numerical difference is still in the range 10^{-2}, whereas it was more in the range 10^{-4} for Darcy case.

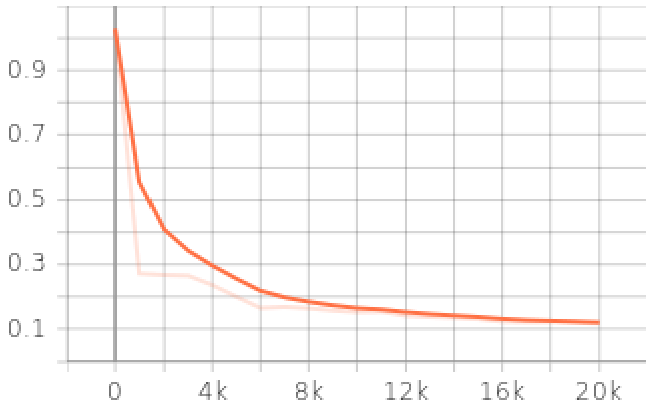


Figure 3: L2 relative error for testing

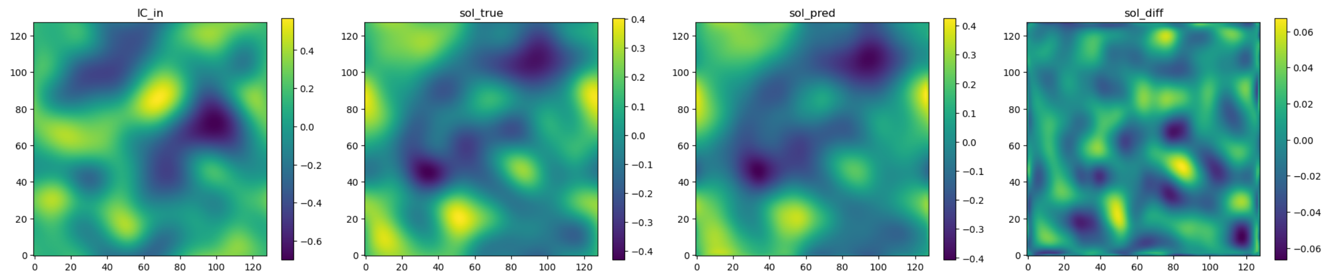


Figure 4: Validators comparison

8 Parametric study

Finally, I tried to modify several parameters in order to see their influence on the error convergence rate and final value.

Firstly, I increased the number of layers for the encoder and the decoder (fig. 5). This action gives a better result for the encoder, and could also a best result for the decoder, but it would need more iterations to achieve it.

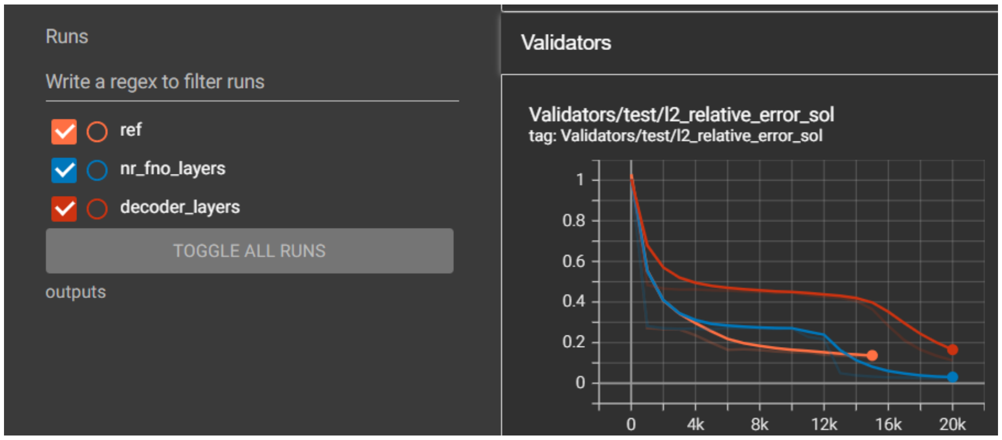


Figure 5: Error behavior regarding the number of layers for the encoder and decoder

Then, I tried to increase the size of the layer of the decoder and to reduce the number of steps of the learning rate (fig. 6). We observe that these two actions give worst results.

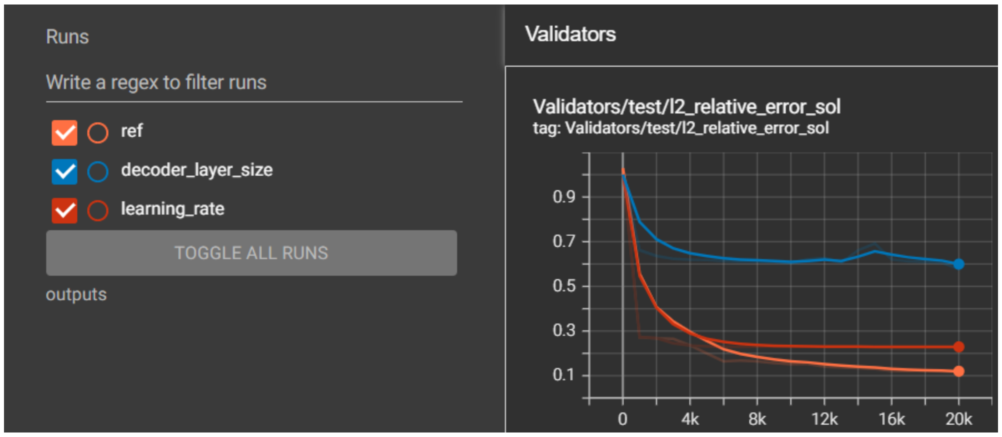


Figure 6: Error behavior regarding the layer size of the decoder and the learning rate