**AI in the sciences and engineering**

**RICCARDO GHETTI**

**TRAINING THE FNO TO SOLVE THE 1D WAVE EQUATION**

1.1) For this task, I used the FNO architecture based on local and global convolutions of the data, where the global convolution is calculated in the Fourier space to simplify computations. This is because we can easily calculate the Fourier transform of functions of which we know values on an evenly spaced grid thanks to the FFT algorithm and because convolution becomes multiplication in the Fourier space. So, at first, I wrote the SpectralConv1d class that actuates the process described earlier: FFT, matrix multiplication cut to the desired number of modes (this is possible because we assume the kernel to be periodic as to have a Fourier series instead of transform), and going back to cartesian coordinates. Then I could write the FNO class, composed of:

* A lifting of the data to higher dimensions: the number of input channels is 2 since I passed to the model the evaluations of the function and the respective grid points. The grid is attached to the data by the get\_grid method. This is done to pass spatial information to the model.
* 3 layers of global and local convolutions which is then transformed by the GELU activation function.
* A projection layer which is a shallow fully connected MLP that outputs 1-dimensional data. This is to allow the processed data to pass information among itself before going back to the physical space.

For the loss function, I directly used the L2 loss between the prediction of the model and the real data, achieving a training error of **1.5** %, and a validation and testing error both around **4.1**%.

1.2) I then calculated the average relative L2 error for the dataset of different resolutions, which forced the number of modes to not exceed 17 since the smallest grid was 32 points.

The error for each dataset is as follows:

32: **8.39**%; 64: **4.41**%; 96: **4.77**%; 128: **5.3**%

The performance is best when using a resolution equal to the one seen during training and the error is remarkably higher for the lower resolution of 32. The inaccuracy also rises for finer and finer resolutions but at a slower pace compared to coarser ones.

1.3) The error between prediction and the out-of-distribution dataset is **8.57**%, higher than any error coming from the given datasets of the same distribution and more than double the one calculated in task1.

1.4) For task4 I defined the FILM layer that performs the following procedure to include information of the time elapsed from the input to the output to the computation:

After every Fourier layer the FILM layer does batch normalization of the data and embeds the time step via a learnable embedding: it scales the data and adds a bias to it derived from said delta t. Skip connections in the convolutions and adding time as an additional input channel seemed to worsen the performance of the model. Also, different activation functions brought different errors overall, with GELU being the best in my experiments.

The training is done on a sample size of 640 since the training dataset is composed of 64 trajectories and 10 possible forward pairs ((0,0.25), (0,0.5) …). Although the training error always converges to 3/4%, the validation error gets stuck at around **30** % on the validation set of snapshots that go from 0s to 1s, which is the remaining trajectories in the train\_sol file. The testing error at the various times is equal to:

t=0.25: **20.9**%; t=0.5: **31.05**%; t=0.75: **38**%; t=1: **29.3**%.

The error is lowest for the predictions at t=0.25 possibly because the time step of 0.25 s is the most numerous in the training set. The second lowest error is associated with prediction at t=1s, maybe because it’s the output snapshot most seen in training. The performance is a lot worse for the intermediate time steps. The error is 4 to 5 times higher than the model for task1. My hypothesis for these big errors is the scarcity of training samples for the model to generalize to different time steps. Other predictions strategies like autoregressive rollouts were very inaccurate with errors arriving at over 200%.

For the out-of-distribution error, I get an error of **35.7**%. The model performance is 5/6% worse than both the testing and validation error, indicating a bigger difficulty to generalize to different types of datasets.

**PDE-FIND: Reconstructing PDEs from data**

My PDE-FIND method roughly operates like this: for a 1+1D data, it first computes the spatial and temporal derivatives through the finite difference method, with custom computation for the extrema of the domain to maintain a second order convergence. Afterwards it flattens the data to construct the theta matrix via the D\_library\_1d function. This method builds a matrix whose columns are powers of the input derivatives up to the specified exponent and then adds the nonlinear terms, composed of multiplications between these derivatives. The degree of these monomials goes from 2 to max\_exp even for the mixed terms. Alongside these computations the function constructs a careful tracking of the description of every column. Subsequently, it finds a sparse solution to the system Ut=Θ(u)\*ξ through the TrainSTRidge algorithm explained in the referenced paper. Although I used the solver given by their repository, I had to update it to be compatible with the modern version of numpy and the given data. Moreover, I changed the train/test split procedure, opting to choose as training data simply the first 80% of the data, remarkably speeding up the convergence of the method. For 2+1D data the function works basically in the same way: computing time and spatial derivatives of both u and v and using all of them to construct the D library, since the supposed equation is coupled. Indeed, the built theta matrix has both the mixed derivates of u and v and their power/ multiplication between them, but it is used to fit both ut and vt separately. In both cases, the function then returns the non-zero elements and their description. In the calculations, I assumed the solution (and numerically verified it) to be smooth enough to ignore the order of differentiation and computed only the non-redundant mixed derivatives.

For the first file the equation seems to be the 1D burger equation with mu=0.1 since the method returns the following (parameters: lambda=1e-4, tol=0.2, max\_exp=4, number of columns of D=47, training tolerance iterations=10):

Ut +1.00098715 u\*ux=0.1002203 uxx

For the second equation the method returns different results based on the l0 penalty and the initial tolerance, but the most consistent terms seem to be u\_xxx and u\*u\_x in some cases and ux (the transport equation) in others even when the tolerance is trained 500 times. The most recurrent found solution was:

Ut+0.0033 uxxx+ 0.0199 u\*ux=0

(parameters: lambda=1e-4, tol=4, max\_exp=2, size of D=15, training tolerance iterations=500)

For the third equation, the computation got significantly longer given the extra dimension, forcing me to choose as max\_exp 1. Choosing 2 or 3 would bring the number of columns of theta to 231 and 631 respectively, bringing the computational time to around 4 hours or more. Therefore, only the linear terms are included in this case. In the end I obtained the following equations (parameters: lambda=1e-4, tol=5, max\_exp=1, size of D=21):

Ut= 0.0553 u+ 0.894 v + 0.0516 uxx + 0.0579 uyy +0.0104vxx – 0.00310272 uxxx

Vt=-0.889 u + 0.0601 v+0.0535 vxx+ 0.0663 vyy

It seems that the first equation isn’t exactly right given the two non-symmetric terms vxx and uxxx, and their low coefficients. The method seems to select them anyway even when training the tolerance more. This may be because of some noise in the first dataset.

With a little bit of effort, it is possible to extend the method to higher dimensional data, but it would lead to very high computational costs and/or the need to subsample a lot, lowering accuracy. In fact, the computational time for the last equation is around 45 minutes, making it impossible to use it on hypothetical 4D or more data. Moreover, it requires some tries to tune the parameters, especially the initial tolerance and the size of D, to obtain a good estimate of the coefficients. The L0 penalty I’d suggest using is 10-4 or 10-5. This phenomenon makes the method hard to trust without any prior knowledge of the process the data comes from. Approaches I didn’t test are other ways of computing derivatives such as smoothing the data with a kernel to eliminate noise or using polynomial approximations, which may possibly make the method more accurate.

**FOUNDATION MODELS FOR PHASE-FIELD DYNAMICS**

For this task, I generated 3 types of initial conditions: Random Fourier series, gaussian mixtures and piecewise continuous functions. I chose a time step of 1/400 seconds meaning the trajectories go from 0s to 0.01s for a total of 5 temporal snapshots. The epsilon I chose for training are: (0.1, 0.075, 0.04, 0.032). This choice is due to the behaviour of the generated data, that goes from smooth evolution to very fast and sharp convergence for the last epsilon. Here is an example for epsilon equal to 0.1 and 0.032 and every type of initial condition:

A graph of a graph

Description automatically generated with medium confidenceA graph of a graph

Description automatically generated with medium confidenceA graph of different colored lines

Description automatically generatedA graph of a function

Description automatically generatedA graph of different colored lines

Description automatically generatedA graph of a number of lines

Description automatically generated with medium confidence

The model consists of the usual FNO architecture and time conditional batch normalization coupled with a similar embedding of the epsilon value without normalization. Training specifications: n\_train=800, n\_test=200, 4 epsilons, loss function=L2 relative loss, training strategy: all2all. Training and validation error after 10 epochs: **6.3**% and **6.5**% respectively. To observe the generalization of the model, I tested it on a dataset with the same types of initial conditions but sharper epsilons (0.025, 0.02). Testing error: **9.52** %. Example of comparison (eps=0.02, Δt=0.005s, IC=Fourier, orange is the prediction)

A graph of a graph

Description automatically generated with medium confidence

Adding the trajectories associated with an epsilon equal to 0.01 brought the average error to 14%, meaning that the model makes a very big average error when generalizing to very fast convergent trajectories. I also tried to test the model on different initial conditions but same epsilons, i.e. higher frequency sinusoids and oscillatory exponentially decaying functions. The periods couldn’t go beyond 20 because an aliasing phenomenon would appear otherwise. The results aren’t satisfactory: around **46%** and **53%** error. Example of comparisons (left: sinusoid, eps=0.04, Δt=0.005s; right: exp. decaying, eps=0.04, Δt=0.0025s):

A graph with blue and orange lines

Description automatically generatedA graph with blue lines and numbers

Description automatically generated

The model seems to not be able to associate high frequency oscillations to smooth trajectories, giving birth to very noisy prediction. Moreover, it tends to overshoot the predictions in areas with higher concavity and can’t handle the extrema of the domain too well. In general, the Allen-Cahn equation is a time dependent pde that makes its solution converge to -1 or 1 in a time interval dependent of epsilon (if the IC have modulus lower than 1). The lower the epsilon, the faster the convergence. Moreover, it is possible for solutions with low epsilons to have a very steep change from one extremum to the other or even more than one if it is highly oscillatory. An example of this is the last picture of both the piecewise and the Fourier series. The solutions always seem smooth.