AISE Assignment 3

Lorenzo Tarricone

June 2024

1 Task1

1.1 Solving the coupled ODE system

To solve the coupled ODE system I used the `diffrax` `jax`library following the structure of the forward solver provided in the documentation example of the library. I define a function `vector_field` that then calls itself the `calculate_force` function. The solver used is a `Tsit5` (fifth-order explicit Runge-Kutta method). The result of the solver dynamics is:

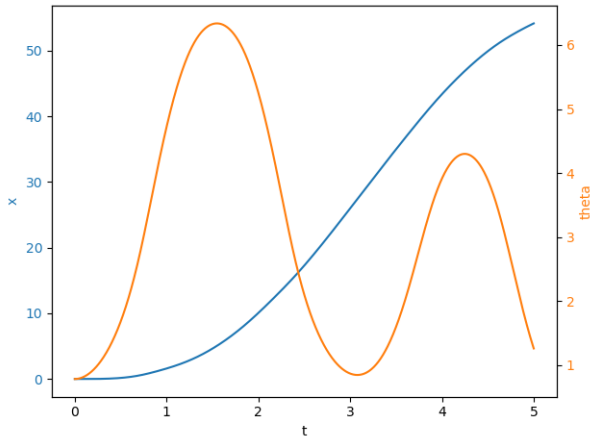

Figure 1: cart position and angle over time

1.2 Learning to balance the pendolum

Here I continued to use the `diffrax` library, but I introduced a NN to model the force $F(t)$ (made by an input, hidden and output layers of respectively 1, 32, 1 nodes). The loss design combines the required criterion for which $\theta(t) \approx 0$ for $t \geq 0.75\ t_N$ (as square deviation from this condition) plus an additional term that imposes penalizes for high values of the force in the second half of

the trajectory. The idea behind this is the fact that we don't want just to move the cart fast enough to arrive fast to the other side of the domain, but we want to balance the pendulum at the beginning of the trajectory and then keep its equilibrium at most giving "small nudges". This penalty was set as half as impactful as the first required one and acts just on the last third of the force trajectory, penalizing quadratically deviations from zero.
The training was done over 3k epochs and produced the following loss curve

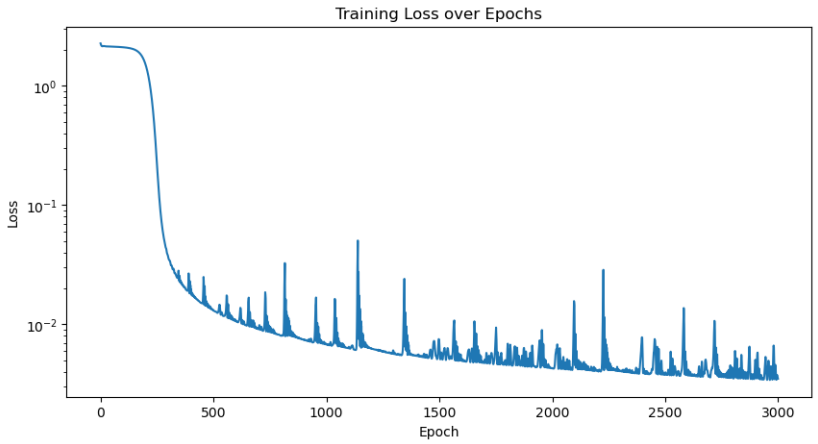

Figure 2: training loss

The desired final result is given in the image below (might be at the end of the report) 3. Please notice how the final angle is 6.286404 radians that renormalized in $[0, 2\pi]$ becomes 0.0032186508 radians

2 Task 2

For all equations to discover I followed a similar strategy.

1. Visualize the data provided
2. Find the size of the mash for each dimension
3. Calculate the derivatives
4. Create a library $\Theta(u)$ with some combination (possibly all, but depends on the complexity of the problem) of the derivatives up to a certain degree
5. Use the functions `STRidge` and `TrainSTRidge` as sparse linear solver to solve for ξ
6. (optional) Redo the process of refining the returned set of coefficients incorporating some intuition of knowledge about the problem (PDE 3) or do post-hoc scoring of subsets of returned non-zero coefficient (PDE 2)

2.1 PDE 1

Here there was no need to do the optional step and the returned solution was Bourger's equation:

$$u_t = 0.1u_{xx} - uu_x$$

(library size 70: all combinations of derivatives up to degree 4)

2.2 PDE 2

This one was a bit more difficult to find and some additional optimization manually was done over all the subsets of the returned elements of the library. The result (adding some suitable regularization) finds the KDV equation with the following coefficients:

$$u_t = -0.1u_{xxx} - 6uu_x$$

(library size 15: all the combinations of derivatives up to degree 2)

2.3 PDE 3

After the first optimization gave a big library I reduced the search space (by trial and error over different categories) by ignoring all the non-linear terms that included partial derivatives. This notably reduced the size of the library and gave a nice and symmetric result of the equation (that matched the plot well, ensuring me of the probable correctness of the result). The final result is:

$$u_t = u + 0.18v + 0.1u_{xx} + 0.1u_{yy} - uuu + 0.78uuv - uvv + 0.78vvv$$

$$v_t = -0.18u + v + 0.1v_{xx} + 0.1v_{yy} - 0.78uuu - uuv - 0.78uvv - vvv$$

(library size first 293 and then 18: described above)

2.4 Issues and possible extensions

I have noticed how the number of library elements included in the second and the third solutions was sensible to the choice of the parameters `lamb` and `d_tol` and I didn't find any systematic way of dealing with these parameters. A possible solution to this ambiguity can be to do something similar to what I have done in this exercise, meaning incorporating knowledge of the solution (by looking at some characteristics of the plot) or playing around with subsets of the returned library. In other words, you can return and score submodels, eventually finding not just one but a family of PDEs that can explain maybe equally well the data. Another issue found was the scaling in computational complexity for the larger grid of the last exercise, which forced me to reduce it by applying some sort of subsampling of the spatial domain.

3 Task Bonus

3.1 Design choices

In order to create the datasets I defined a custom function that creates the datasets relying on existing libraries (e.g. `umpy.polynomial.chebyshev` and `sklearn.gaussian_process`) coefficient for slopes/intercepts PL and for coefficients CP were sampled uniformly at random in (-3, 3).

To solve the 1D Poisson equation with the required boundary conditions I have used the dedicated function in `pypde`. All the data (divided into training and test for each class) were stored as `torch.Tensors` in a dictionary.

As a neural operator, I have taken the `1dFNO` from the tutorials and it was trained initially for 100 epochs. The finetuning on different datasets was done on just 30 epochs on the 20 examples randomly taken from the test set.

3.2 Discussion of the results

The results obtained are the following:

Trained on	Eevaluated on	Zero-shot loss	Finetuned loss	relative L2 error (%)
GP	GP	0.054	0.054	-0.154
GP	PL	0.14	0.035	75.386
GP	CP	0.369	0.093	74.673
PL	GP	0.062	0.054	12.034
PL	PL	0.032	0.032	0.576
PL	CP	0.082	0.082	0.138
CP	GP	0.054	0.055	-1.757
CP	PL	0.037	0.032	13.929
CP	CP	0.075	0.078	-3.465

Some things are notable from these results:

1. Fine-tuning on the same type of functions that the model was trained on gives no significant improvement (with all values around zero and the one for CP even around -3%)
2. Finetuning the models trained on GP really improves their performances, maybe indicating (counterinutitively) that the type of functions modelled by a GP are significantly different from the ones modelled by a PL function or a CP
3. It doesn't look like training on some specific class brings a particular advantage. Maybe by increasing the number of datapoints sampled and the number of those included for the finetuning would reveal additional patterns.

3.3 Use of PINNs

In this exercise, I identify two possible uses of PINNs

1. **Solving the 1D Poisson Equation**: In this case, I would stick with a traditional forward solver as we have seen how in general (especially for simple functional forms like the one considered in this case) these methods are much faster as they don't require any training of the model.

2. **Using it to learn the mapping** $f(x) \rightarrow u(x)$: Here I would also continue to use a Neural Operator instead of a PINN because of its ability to learn the mapping between functions spaces that are independent on the mesh-size (which was the main reason we introduced the topic in class).

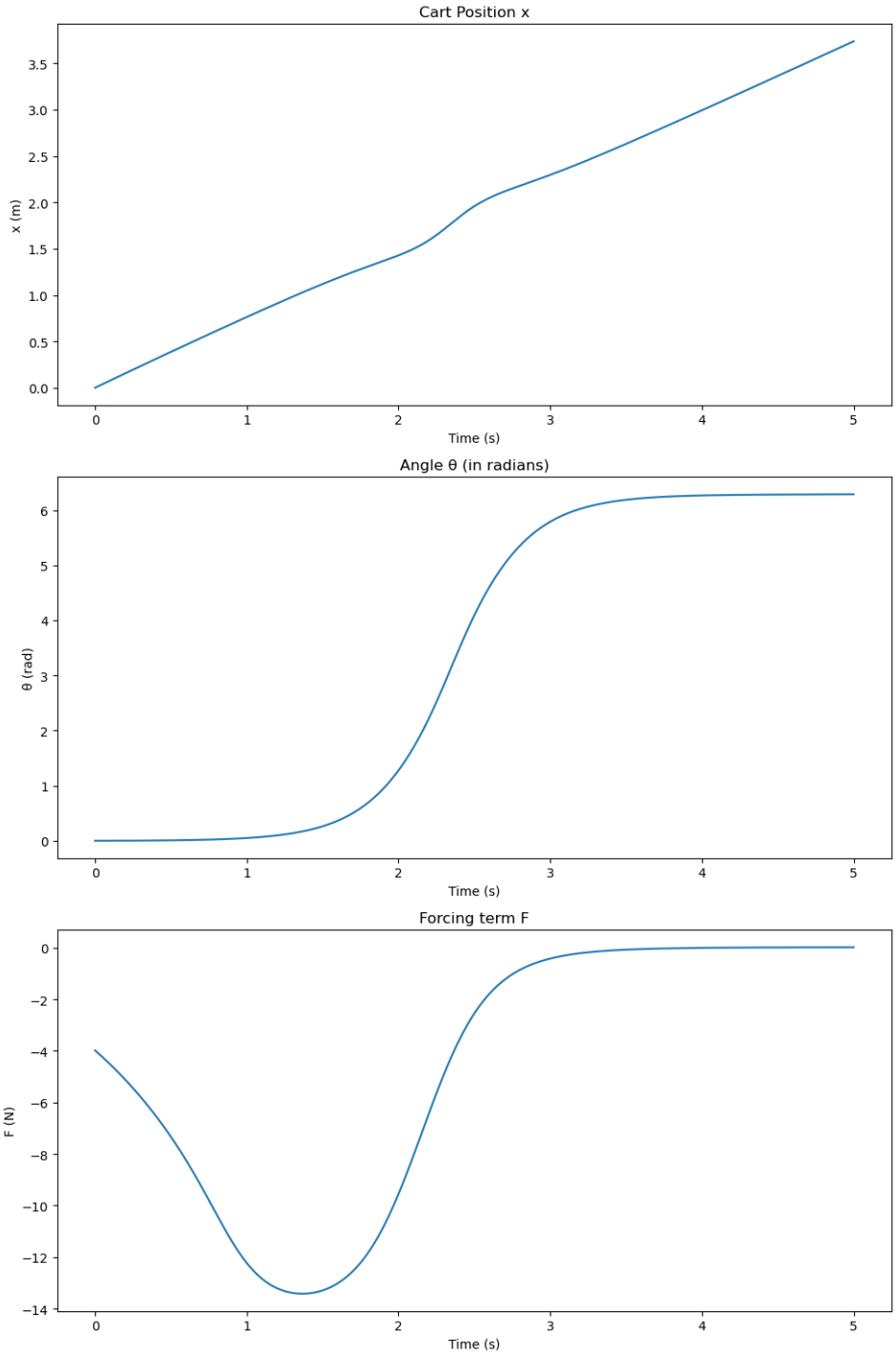

Figure 3: desired variables over time