Neural network solving BVP

We are considering the problem of approximating a curve

$$[0, L] \ni \bar{s} \mapsto q(\bar{s}) \in \mathbb{R}^2.$$

This curve is known to satisfy a certain differential equation and the related boundary conditions. We consider the following abstract setting

$$\begin{gathered}\text{Find } q(\bar{s}) \text{ such that } \mathcal{L}\left(q(\bar{s}), q'(\bar{s}), q''(\bar{s})\right) = 0 \\ \text{and } q(0) = q_1, q(L) = q_N, q'(0) = v_1, q'(L) = v_N.\end{gathered}$$

We decide to parametrise an approximation $q_\theta(s)$, $\frac{\bar{s}}{L} = s$ of $q(\bar{s})$ in a differentiable way, and then train the parameters of $q_\theta(s)$ to satisfy some conditions characterising the true curve $q(\bar{s})$.

Because of the importance of the boundary conditions in BVPs, we decide to construct $q_\theta(s)$ so it satisfies those 4 conditions regardless the choice of the free parameters in the model. To do so, we define

$$q_\theta(s) = \mathcal{N}_\theta(s, q_1, q_N, v_1, v_N) + \sum_{i=0}^{3} a_i(\theta) s^i$$

where $\mathcal{N}_\theta : \mathbb{R} \to \mathbb{R}^2$ is a generic neural network depending on the parameter $\theta \in \Theta$, and $a_1, ..., a_3 : \Theta \to \mathbb{R}^2$ are functions adapted to $\mathcal{N}_\theta$ that allow to get

$$q_\theta(0) = q_1,\ q_\theta(1) = q_N,\ q'_\theta(0) = v_1,\ q'_\theta(1) = v_N.$$

We remark that we choose to add a cubic polynomial because it is enough to impose 4 conditions. Indeed, we have

$$q_\theta(0) = \mathcal{N}_\theta(0, q_1, q_N, v_1, v_N) + a_0(\theta) = q_0 \implies a_0(\theta) = q_0 - \mathcal{N}_\theta(0, q_1, q_N, v_1, v_N)$$

and the other conditions can be found using Maple and amount to

$$\begin{aligned}a_1(\theta) &= v_1 - \left.\frac{d}{ds}\mathcal{N}_\theta(s)\right|_{s=0} \\ a_2(\theta) &= 2\left.\frac{d}{ds}\mathcal{N}_\theta(s)\right|_{s=0} + \left.\frac{d}{ds}\mathcal{N}_\theta(s)\right|_{s=1} - 3\left(\mathcal{N}_\theta(1) - \mathcal{N}_\theta(0)\right) \\ &\quad +3\left(q_N - q_1\right) - 2v_1 - v_N \\ a_3(\theta) &= -\left.\frac{d}{ds}\mathcal{N}_\theta(s)\right|_{s=0} - \left.\frac{d}{ds}\mathcal{N}_\theta(s)\right|_{s=1} + 2\left(\mathcal{N}_\theta(1) - \mathcal{N}_\theta(0)\right) \\ &\quad +2\left(q_1 - q_N\right) + v_1 + v_N,\end{aligned}$$

where for compactness we denoted with $\mathcal{N}_\theta(s)$ the value $\mathcal{N}_\theta(s, q_1, q_N, v_1, v_N)$.

Given this parametrisation, there are multiple ways one can find the parameters $\theta \in \Theta$. We first mention them and then point out their pros and cons, especially comparing this parametrisation strategy with other ones.

Fully supervised regression task

One way to find the parameters is to use available discretisations of the solution curve $q(s)$, and possibly of its tangent vector $q'(s)$. This dataset thus amount to the pairs

$$\mathcal{T} = \left\{ \left(s_i = \frac{ih}{L}, q_1, q_N, v_1, v_N \right), q_i(q_1, q_N, v_1, v_N) \right\}_{i=0}^{N},$$

where q_i is the solution of the BVP on the node i, and satisfying the boundary conditions (q_1, q_N, v_1, v_N). In this case we consider a curve (beam) of length L, where $h = \frac{L}{N}$ is the length of one element of the discretisation, and $N+1$ is the number of the nodes.

This naturally leads to a fully supervised regression task, where we look for a parameter set θ minimising

$$\text{Loss}(\theta) = \frac{1}{N} \sum_{(q_1, q_N, v_1, v_N) \in \mathcal{B}} \sum_{i=1}^{N} \| q_\theta(s_i, q_1, q_N, v_1, v_N) - q_i(q_1, q_N, v_1 . v_N) \|^2,$$

where $\mathcal{B}$ is the set of available boundary conditions.

We notice that here we are not directly using the fact that $q_\theta(s, q_1, q_N, v_1, v_N)$ is a continuous function of s. In other words, with the dataset $\mathcal{T}$ it is also possible to directly approximate the discretisation of the solution q, i.e. to work with a neural network defined as

$$\overline{\mathcal{N}}_\theta\left(q_1, q_N, v_1, v_N\right) = \begin{bmatrix} q_1 \\ \mathcal{N}_\theta^1\left(q_1, q_N, v_1, v_N\right) \\ q_N \\ v_1 \\ \mathcal{N}_\theta^2\left(q_1, q_N, v_1, v_N\right) \\ v_N \end{bmatrix} \in \mathbb{R}^{4(N+1)},$$

where $\mathcal{N}'_\theta : \mathbb{R}^8 \to \mathbb{R}^{2(N-1)}$, $i = 1, 2$. Theoretically both options are viable, however there are a couple of possible drawbacks of this second approach:

- It is not independent on the spatial discretisation provided by the training data. In other words, if more than one finite element discretisation is provided, this method can not exploit them since it is constrained to the specific number of discretisation nodes.

- It is reasonable to desire that the quality of the approximation provided by the parametric model improves when the number of discretisation nodes increases, as in classical finite element theory. However, this discrete approximation would probably suffer from this increase since the vector valued functions to approximate becomes very high dimensional.

This suggests that even with the choice of this fully supervised training strategy, having a smooth function of the parameter s can be beneficial. Indeed, we remark that this strategy would also be compatible with different discretisations of the curve (beam) and would not suffer from the curse of dimensionality in case more nodes are added.

Implementation

This is the solution which is currently implemented in our code [4]. Such implementation includes a few scripts to test the hyperparameters, and see what is the best performing architecture. These scripts are based on Optuna, see [5].

The current implementation allows to choose if to impose the boundary conditions by design, adding the aforementioned cubic polynomial, or to only train an unconstrained neural network. This is possible by choosing either *correct_functional=True* (to add the polynomial), or to *False* otherwise.

The code is supported by a few comments and a readme in the repository. The current implementation has probably an issue with the computation of the derivatives, since even if the computed approximation is accurate, the associated derivatives seem to not be so. The derivative is implemented with the forward mode of automatic differentiation, thanks to the method *jacfwd* of torch.functional. An example of such method is provided in the repository.

Residual based learning

Since the approximate curve $q_\theta(s)$ should reproduce the behaviour of the solution to the differential problem

$$\mathcal{L}\left(q(s), q'(s), q''(s)\right) = 0,$$

another possibility to find θ is to minimise the discrepancy

$$\| \mathcal{L}\left(q_\theta(s), q'_\theta(s), q''_\theta(s)\right) \|$$

in sufficiently many locations $s_1, ..., s_M \in [0, 1]$. The derivatives can easily be computed with automatic differentiation, and one could hence optimise the following loss funciton

$$\text{Loss}(\theta) = \frac{1}{M} \sum_{k=1}^{M} \| \mathcal{L}\left(q_\theta\left(s_k\right), q'_\theta\left(s_k\right), q''_\theta\left(s_k\right)\right\|^2 .$$

Also in this case, with $q_\theta(s_k)$ we mean $q_\theta(s_k, q_1, q_N, v_1, v_N)$.

The remarkable aspect of this strategy is that it is completely unsupervised, so it does not require any training data. Furthermore, it is more likely that the approximation is physically meaningful since it is derived following a variational approach. On the other hand, as various other papers point out, it might be hard to get a good appromation with M relatively small.

Mixed approach

Various papers suggest that combining residual based learning techniques with some available data can considerably improve the accuracy of the approximation. A natural way to do so is by adding two contributions in the loss function, which hence becomes

$$\text{Loss}(\theta) = \underbrace{\frac{\gamma_1}{N} \sum_{(q_1, q_N, v_1, v_N) \in \mathcal{B}} \sum_{i=1}^{N} \left\| q_\theta\left(s_i, q_1, q_N, v_1, v_N\right)\right) - q_i(q_1, q_N, v_1, v_N) \|^2}_{\text{data fitting}} + \underbrace{\frac{\gamma_2}{M} \sum_{k=1}^{M} \left\| \mathcal{L}\left(q_\theta\left(s_k\right), q'_\theta\left(s_k\right), q''_\theta\left(s_k\right)\right) \right\|^2}_{\text{Physical regularisation}}.$$

Here we suppose

$$\mathcal{T} = \{(s_i, q_i)\}_{i=1}^N$$

is a set of available datapoints, and γ_1, γ_2 are two hyperparameters weighting the data-fitting part and the physical regularisation term. To optimise this multi-objective loss function it might be important to fine tune (i.e. balance) the two weights γ_1, γ_2, and here there are various options. An interesting one can be found in [2], and it is implemented in [3].

Useful references

[1] Mortari, D. (2017). The theory of connections: Connecting points. *Mathematics*, *5*(4), 57.

[2] Heydari, A. Ali, Craig A. Thompson, and Asif Mehmood. "Softadapt: Techniques for adaptive loss weighting of neural networks with multi-part loss functions." *arXiv preprint arXiv:1912.12355* (2019).

[3] GitHub - dr-aheydari/SoftAdapt: Implementation of the SoftAdapt paper (techniques for adaptive loss balancing of multi-tasking neural networks)

[4] davidemurari/elastica · GitHub

[5] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In KDD