Homework 8
APPM 4720/5720 Scientific Machine Learning, Fall 2024

Due date: Monday, Oct. 28 '24, before midnight, via Gradescope

Instructor: Prof. Becker

Revision date: 10/22/2024

Theme: PINNs in "inverse mode"

Instructions Collaboration with your fellow students is OK and in fact recommended, although direct copying is not allowed. The internet **is allowed for basic tasks** (e.g., looking up definitions on wikipedia, looking at documentation, looking at basic tutorials) but it is not permissible to search for solutions to the exact problem or to *post* requests for help on forums such as `http://math.stackexchange.com/`.

Setup Like lab 8, we're looking at the logistic differential equation, to find a function $u : [0, T] \to \mathbb{R}$ such that

$$u'(t) = \theta u(t)\,(1 - u(t)/k) \tag{1}$$

for a carrying capacity k (we'll take $k = 10$ to be fixed for the entire homework), a given growth rate θ, and an initial condition $y(0) = y_0$. We'll also fix $T = 3$ for the entire homework.

Problem 1: **Warmup**: numerically solve the logistic ODE using $y_0 = 1$ and parameter $\theta = \pi$ on the interval $[0, T = 3]$ (e.g., using `scipy.integrate.solve_ivp`). Turn in your code and a plot of u. Save data on the solution at 21 equally spaced points, e.g., $\{(t_i, u(t_i))\}_{i=1}^{21}$ to be used in the next problem.

Problem 2: Now we'll suppose we have the dataset $\{(t_i, u(t_i))\}_{i=1}^{21}$ and we know u satisfies the ODE from Eq. (1), but we *don't know* θ. Solve for θ using the idea of PINNs (i.e., have two parts to your loss function, one for satisfying the ODE, and one for satisfying the data constraint; and since the ODE is not fully known, you'll have θ be a "parameter" and include it in the gradient step), using an initial guess of $\theta = 2$. Turn in your code, a plot of your PINN solution, and a short discussion of what value θ you found, and if you ran into any difficulties along the way (i.e., if you don't get it to work, discuss why, so that you get partial credit!).

Tips There are different ways to program this in PyTorch. Below are some of the tips I used when I solved it; you may or may not need to use these depending on your setup.

- I used the SIREN architecture that we've used before in labs and homeworks, and 3 to 4 hidden layers with 30 to 60 parameters seemed reasonable. If you want to find the smallest possible network that is still expressive enough, you can "cheat" and use a pure regression (or "implicit neural compression" INR approach), where you use samples $\{(t_i, u(t_i))\}_{i=1}^{n}$ except use $n \approx 10^3, 10^4$. This is simpler code so fewer bugs, and you can see if your architecture can fit the data.
- With the ODE constraint, if the output of u is outside the range $[0, k]$, this is unphysical and u' can be very large. I found it helpful to modify my SIREN architecture to project the output to be in the range $[0, k]$. You can use `torch.clamp` for this.
  - It's possible the clamping has made the default SIREN initializations not work well, or maybe it's something else, but I found my PINN to be very sensitive to initial conditions. Sometimes it failed, but then re-running it with a new realization of initial conditions worked fine. So make sure to try re-running if it doesn't work at first.

- I used L-BFGS (typically good results in 100 epochs, better results in 500 epochs) with 1000 collocation points (you can choose the number of collocation points; on the other hand, the number of data points is fixed at 21).
  - I preferred L-BFGS over SGD because it will adapt better if the parameters are of different scales (which θ might be) and because it converges quickly (the main reasons *not* to use it is if the problem is too large, but ours is not that large).
  - Adjusting the hyperparameters of L-BFGS makes a difference. I used a learning rate of 1 and `history_size=20, max_iter=5, line_search_fn='strong_wolfe'`. Allowing more `max_iters` (which controls the line search iterations) often results in bad performance in my experience.
- You can either use a fixed set of collocation points (e.g., `torch.utils.data.TensorDataset`) or always draw a new random batch (e.g., `torch.utils.data.IterableDataset`). Both worked OK for me. You don't have to use the PyTorch dataloader framework if you don't want to; if you do, then it takes a bit more programming work to setup the `IterableDataset` version.
- You need to train θ. One way to do this is to make it a parameter in your neural net model, and then you can automatically train it in the optimizer using the usual `torch.optim.LBFGS(model.parameters(),...)` step (or `torch.optim.SGD(model.parameters(),...)` etc.). To do this, you need to register θ as a "parameter" in the model. You can do this inside your `nn.Module` model class as `self.theta = torch.nn.parameter.Parameter( ... initial-value-of-theta ... )`.
  - To start with, keep θ fixed at either the true value or close to the true value (since you actually know it!). This will help you find other bugs in your code. Once that works, then allow θ to vary.
  - The scale of θ may be very different than the scale of the weights, which could lead to it not moving very fast during the gradient steps. One fix is to rescale θ with a change-of-variables. I found that scaling it by a factor of about 100 to 1000 was critical to good performance..
- As always with PyTorch, you may need to reshape tensors to be the right size. The first dimension should always be the batch dimension. You can convert numpy arrays to the right torch object by doing `torch.tensor( ... ).to(torch.float32)` (and possibly reshape, e.g., `.unsqueeze(1)` or similar).
- Once I had reasonable hyperparameters, my PINN trains on my laptop (no GPU) in the range of 1 to 10 minutes, depending on the exact hyperparameters and number of epochs (and for some hyperparameters, I can get reasonable results in a handful of seconds if we have a lucky initialization). So if your code is always running > 10 minutes, try changing something.