Chapter 2

Project 1: Solving differential equations with neural networks

2.1 An introduction to scientific machine learnings

Scientific machine learning is an emerging discipline within the mathematical sciences, which is concerned with solving problems in real-world applications that have traditionally been solved using classical computational methods. What sets scientific machine learning apart from general machine learning and generative artificial intelligence is the inductive bias provided by the physical laws of the universe that should be accurately reflected in all developed models in this field. An example for such an inductive bias could be conservation of energy in a machine learning based model for the motions of the planets around the sun, or the preservation of angular momentum of two bodies rotating around one another. While modern scientific machine learning emerged less than 10 years ago, it has developed into a prolific research discipline at the intersection of mathematics, computer science, physics and engineering.

Most models from the mathematical sciences are described in the form of differential equations, and as such developing machine learning methods that can include differential equations in some appropriate form is one of the main areas of scientific machine learning. Specifically, recent years have seen a surge of interest in so-called *physics-informed machine learning*, which is a subfield of scientific machine learning, devoted to solving differential equations using machine learning rather than using classical scientific computing. In this project we will investigating physics-informed neural networks, which are neural networks that are trained to approximate solutions of differential equations.

2.2 An introduction to physics-informed neural networks

Differential equations are traditionally solved with classical numerical methods such as finite differences, finite volumes or finite element methods. Recent years have seen the emergence of an alternative solution strategy based on neural networks. The solution of differential equations with neural networks was first proposed in the late 1990s [6] and, as most research on neural networks, has seen an explosion of interest in recent years with the advent of affordable GPU computing and availability of computational frameworks such as `JAX`, `TensorFlow` and `PyTorch`. Today, the method originally proposed in [6] is referred to as *physics-informed neural networks* (PINNs), see [9]. Physics-informed neural networks, along with various extensions to operator learning, are one of the main areas of scientific machine learning today.

To introduce the idea behing PINNs, we begin with a high-level overview. In standard deep learning one trains a neural network solely based on data with the goal to either learn an input–output mapping (supervised learning), or to learn an underlying data distribution (unsupervised learning). If the data that is to be learned is known to come from a differential equation (for example recording the motion of the planets around the sun, which follows the laws of classical and/or relativistic mechanics), then one should aim to train a neural network not solely based on data but rather in a manner that also enforces the underlying differential equations. Including important extra information into a machine learning algorithm is referred to as an *inductive bias*. Knowing the appropriate inductive bias for a problem is typically critical as it often leads to better numerical results compared to generic machine learning algorithms. Thus, for physics-informed neural networks, the inductive bias is the given differential equation itself, and a neural network is then being trained to approximate the solution of this differential equation, along with any given initial and/or boundary conditions for that differential equation.

In practice, the neural network being learned accepts as input the independent variables of the differential equations and outputs the dependent variables. Training of this neural network is done by enforcing that *(i)* the differential equations, *(ii)* the initial condition, and *(iii)* the boundary conditions hold on finitely many collocation points sampled (typically at random) over the domain of the independent variables[1]. The neural network learns to enforce these constraints by minimizing a suitable loss function that consists of mean-squared errors of these three contributions. Hence, physics-informed neural networks are a competing approach for solving differential equations that bypasses any numerical discretization of the differential equations at hand.

2.3 Physics-informed neural networks for a single ODE

Suppose we are given the simple decay problem

$$\frac{\mathrm{d}u}{\mathrm{d}t} = -u(t),$$

with the initial condition $u(t=0) = u_0$. We know that the analytical solution to this problem is $u(t) = u_0 e^{-t}$. We now aim to solve this problem using physics-informed neural networks. That is, we aim to train a neural network $\mathcal{N}^{\boldsymbol{\theta}}(t)$ such that the true solution $u(t)$ is approximated by the numerical solution $u^{\boldsymbol{\theta}}(t) = \mathcal{N}^{\boldsymbol{\theta}}(t)$, i.e. $u(t) \approx u^{\boldsymbol{\theta}}(t)$. The associated neural network $\mathcal{N}^{\boldsymbol{\theta}}$ thus has to accept a single scalar input (the time t) and produce a single scalar output (the numerical solution $u^{\boldsymbol{\theta}}$). Note that the quality of the numerical solution will depend on the weights and biases $\boldsymbol{\theta}$ of the neural network $\mathcal{N}^{\boldsymbol{\theta}}$. The class of neural networks typically used for PINNs are standard multi-layer perceptrons, using a suitable (differentiable!) activation function throughout all layers of the network. Other neural network architectures (e.g. convolutional neural networks [3], transformer-based architectures [10] or Kolmogorov–Arnold networks [7]) can be used as well.

To train the neural network $\mathcal{N}^{\boldsymbol{\theta}}$ to indeed solve the given differential equation, its loss function has to incorporate the associated initial value problem. Let us define the *physics-informed loss function*

$$\mathcal{L}(\boldsymbol{\theta}) \sim \mathcal{L}_\Delta(\boldsymbol{\theta}) + \gamma \mathcal{L}_\mathrm{i}(\boldsymbol{\theta}),$$

[1] We make all of this precise below.

where

$$\mathcal{L}_\Delta(\boldsymbol{\theta}) \sim \left(\frac{\mathrm{d}u^{\boldsymbol{\theta}}}{\mathrm{d}t} + u^{\boldsymbol{\theta}} \right)^2$$

is the *differential equation loss* and

$$\mathcal{L}_\mathrm{i}(\boldsymbol{\theta}) = \left(u^{\boldsymbol{\theta}}(0) - u_0 \right)^2$$

is the *initial value loss*, with γ being a positive constant. If $\mathcal{L}_\Delta = 0$ the differential equation will be exactly satisfied and if $\mathcal{L}_\mathrm{i}(\boldsymbol{\theta}) = 0$ the initial value will be exactly satisfied by the neural network solution $u^{\boldsymbol{\theta}}$. Note that the loss weight constant γ is important as there is no a priori guarantee that the magnitudes of the differential equation and the initial value losses will be comparable. If one of the two loss contributions dominates the total loss then choosing an appropriate value for γ can be essential to make sure that both loss components contribute equally to the total loss.

A novel aspect in this problem is that the loss function $\mathcal{L}$ explicitly depends on the derivative of $u^{\boldsymbol{\theta}}$. How can we compute this derivative? Fortunately, we can use the same automatic differentiation routine for *exactly* (up to machine precision) computing this derivative that is used to compute the gradient of the neural network with respect to the weights of the network. This works as a neural network is a differentiable function of its inputs (hence why differentiable activation functions are required), the derivatives of which can be obtained exactly with automatic differentiation. Thus, no numerical differentiation such as finite differences or Runge–Kutta time stepping is necessary, which strongly simplifies the discretization procedure in comparison to traditional numerical methods. Physics-informed neural networks are thus truly meshless methods, which allows using them on complicated domains, such as on embedded surfaces or for problems with complicated boundary conditions.

Here is a code snippet illustrating how the PINN loss can be computed in `TensorFlow`:

```
# Outer gradient for tuning network parameters
with tf.GradientTape() as tape:

  # Inner gradient for derivatives of u w.r.t. t
  with tf.GradientTape() as tape2:
    tape2.watch(t)
    u = model(t)

  # Derivative of the neural network solution
  ut = tape2.gradient(u, t)

  # Define the differential equation loss
  eqn = ut + u
  DEloss = tf.reduce_mean(eqn**2)

  # Define the initial value loss
  u_init_pred = model(t0)
  IVloss = tf.reduce_mean((u_init_pred - u_init)**2)

  # Composite loss function
  loss = DEloss + gamma*IVloss

# Return the model gradients for the gradient descent step
grads = tape.gradient(loss, model.trainable_variables)
```

One important detail we have omitted so far is *where* we want to evaluate/minimize the loss function. The initial value problem considered is defined over the temporal domain $[0, t_\mathrm{f}]$, so we want to evaluate our loss function for that domain only. Unfortunately, it would not be feasible to minimize the loss everywhere in $[0, t_\mathrm{f}]$, which contains infinitely many points! Instead, we sample our temporal domain at finitely many random locations t_i, $i = 1, \ldots, n$, i.e. we have $0 \leqslant t_i \leqslant t_\mathrm{f}$ for each $i = 1, \ldots, n$. We then define the differential equation loss at these finitely many points only,

$$\mathcal{L}_\Delta(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{\mathrm{d}u^{\boldsymbol{\theta}}(t_i)}{\mathrm{d}t} + u^{\boldsymbol{\theta}}(t_i) \right)^2 .$$

If this loss function is zero, then the neural network will exactly satisfy the differential equations in the points t_i.[2] The actual composite physics-informed loss function being minimized is therefore the mean squared error loss

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{\mathrm{d}u^{\boldsymbol{\theta}}(t_i)}{\mathrm{d}t} + u^{\boldsymbol{\theta}}(t_i) \right)^2 + \gamma (u^{\boldsymbol{\theta}}(0) - u_0)^2 .$$

The points $\{t_i\}|_{i=1,\ldots,n}$ are referred to as *collocation points*. They are akin to the time steps in a numerical solution of differential equations. Note that these collocation points could be chosen arbitrarily. They could lie on a regular mesh, etc., but it was found experimentally [9] that sampling them using suitable random sampling, such as Latin hypercube sampling, gives the most accurate numerical results. For stiff problems or for partial differential equations that can develop shocks or rapidly changing solutions, it was also found beneficial to adaptively refine the collocation points used throughout the (spatio-)temporal domain of the problem.

For our initial value problem defined over $[0, t_\mathrm{f}]$, for the example of the final time $t_\mathrm{f} = 1$, here is how 20 collocations points could be randomly distributed:

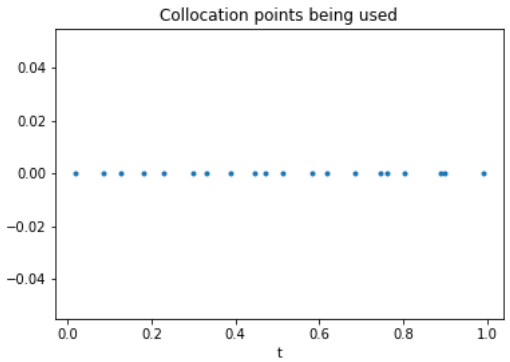

Figure 2.1: An example for a collection of 20 collocation points, randomly sampled from the interval [0,1]. In each of the collocation points, we aim that the given MLP approximation for u satisfies the differential equation $u' = u$.

With all of this in place, we can solve our differential equation using a neural network. We choose here an MLP with 4 hidden layers and 20 units each, using the hyperbolic tangent activation function, to obtain the following solution, using $n = 100$ collocation points:

[2] This is akin to Hermite interpolation, with the enforcement of derivatives at the interpolating points being replaced with enforcing that a differential equation holds for a function parameterized by a neural network. You will see/might have seen Hermite interpolation in MATH 3132.

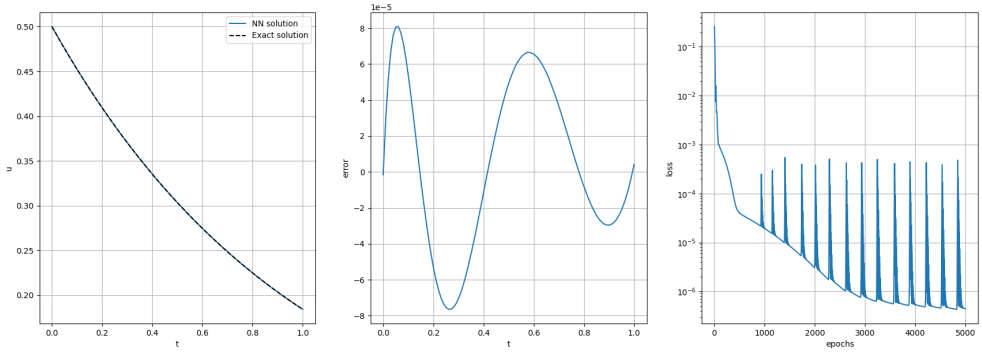

Figure 2.2: Results for a trained physics-informed neural network for the decay problem $u' = u$, with initial condition $u(0) = 1$. *Left:* Solution obtained from the physics-informed neural network. *Middle:* Numerical error compared to the exact solution $u(t) = e^{-t}$. *Right:* Loss as a function of the number of training steps.

2.4 Project

This section contains a concise description of the tasks to be complete for this module.

2.4.1 Physics-informed neural networks for systems of differential equations

The code used to train the physics-informed neural network shown in Fig. 2.2 is provided on `Brightspace`. The goal of this first part of this module is to extend this code to a system of differential equations. In particular, we consider the Lorenz–1960 model, which was derived in [8] as the minimal model that retains some relevance for the equations governing the evolution of the atmosphere. This model is given as

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= kl\left(\frac{1}{k^2+l^2} - \frac{1}{k^2}\right) yz, \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= kl\left(\frac{1}{l^2} - \frac{1}{k^2+l^2}\right) xz, \\ \frac{\mathrm{d}z}{\mathrm{d}t} &= \frac{kl}{2}\left(\frac{1}{k^2} - \frac{1}{l^2}\right) xy, \end{aligned} \tag{2.1}$$

with initial conditions $x(0) = x_0$, $y(0) = y_0$ and $z(0) = z_0$. Here k and l are positive constants.

Complete the following tasks:

1. Extend the provided code on `Brightspace` for solving the simple decay problem to solving the Lorenz–1960 model instead. The goal is to solve the Lorenz–1960 model for a variety of final integration times t_f. In particular, experiment with $t_\mathrm{f} \in \{1, 5, 10, 20\}$, using a suitable number of collocation points n and then training the physics-informed neural network until the loss starts to level out. Choose $k = 1$ and $l = 2$, but feel also free to experiment with different values for k and l to see how your results change. As initial conditions, use $x(0) = 1$, $y(0) = 0.5$ and $z(0) = 1$.
2. In addition to the time series of the total loss (shown in Figure 2.2), also plot the time series for the individual loss components $\mathcal{L}_\Delta$ and $\mathcal{L}_\mathrm{i}$. Also note that for the Lorenz–1960 model, the differential equation and the initial condition loss each consist of three components,

one for each variable x, y and z and so multiple loss weight constants γ might be needed. Which of the loss components dominates the total loss? Does this inform how you should scale each loss component with an appropriate value for the γ?

3. To compare your results against another numerical method, use `scipy`'s `solve_ivp` to obtain a numerical solution using a classical Runge–Kutta integrator. How large is the discrepancy, i.e. the error, between the physics-informed neural network and the Runge–Kutta integration for different t_f?

Remark 3. Physics-informed neural networks requiring solving what is referred to as a *multi-task* optimization problem. There is one task associated with each differential equation and initial condition, and adding all the loss components associated with each task together in a weighted sum is just one of the many ways how multi-task optimization problems can be tackled[3]. This is still a field of active research, both for physics-informed neural networks and for general multi-task optimization problems arising in machine learning research.

2.4.2 Physics-informed neural networks for higher-order equations

In the previous task we have extended physics-informed neural networks to systems of first order differential equations. In this task, we investigate the possibility to extend them to higher-order differential equations.

You may have seen in your ODEs class that any nth order differential equation can be equivalently written as a system of n first-order equations. Incidentally, this is what you have to do to use off the shelf numerical integrators such as `scipy`'s `solve_ivp`, which only works for systems of first order differential equations. However, for physics-informed neural networks a challenge that arises is that an nth order differential equation requires just a single differential equation loss term while the associated system of n first order equations would require n loss terms. As such, the associated optimization problem to be solved can be harder as it might be more challenging to balance and minimize n loss contributions rather than a single loss contribution[4]. This is one of the many ways in which scientific machine learning differs from classical scientific computing.

Here we study the simple harmonic oscillator, a single second-order ordinary differential equation given by

$$m\frac{\mathrm{d}^2u}{\mathrm{d}t^2} + ku = mu'' + ku = 0, \tag{2.2}$$

where m and k are the mass and the spring constant of the harmonic oscillator, respectively. The initial conditions are given by $u(0) = u_0$ and $u'(0) = u_0'$. We choose $u_0 = u_0' = 1$ and $m = 1$ and $k = 2$. For the final integration time, choose 1, 2 and 5 full periods of the solution.

Complete the following tasks:

1. Solve the harmonic oscillator with a physics-informed neural network by representing (2.2) as a system of two first-order differential equations.
2. Solve the harmonic oscillator with a physics-informed neural network by solving (2.2) directly as a second-order equation.

Which of the two representations gives better numerical results?

[3] If you want to learn more about this, you can do a literature search of Pareto optimality.

[4] Of course, both representations of the nth order equation would still require n initial or boundary conditions.

2.4.3 Improving physics-informed neural networks with hard constraints

One of the main drawbacks of the above approach to physics-informed neural networks is that it requires one to learn the initial condition. This is unsatisfactory as, in contrast to the solution of the differential equation itself, the initial condition is known and therefore should not have to be learned by the neural network.

It is possible to directly include the initial condition into the solution ansatz for the neural network solution as a *hard constraint*. Hard constraining the initial condition means that the neural network solution will satisfy the initial condition for all values of the neural network weights.

There are infinitely many ways for hard-constraining an initial condition. For the single ordinary differential equation $u' = f(t, u)$ with initial condition $u(0) = u_0$, a simple hard constrained neural network is

$$u^{\boldsymbol{\theta}}(t) = u_0 + t \cdot \mathcal{N}^{\boldsymbol{\theta}}(t). \tag{2.3}$$

Note that for all values of the weights $\boldsymbol{\theta}$ of the neural network $N^{\boldsymbol{\theta}}(t)$ we necessarily have $u^{\boldsymbol{\theta}}(0) = u_0$ and as such we do not have to enforce the initial condition during training of the physics-informed neural network. This means, that the total physics-informed loss reduces to the differential equation loss, i.e. $\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_\Delta(\boldsymbol{\theta})$.

Complete the following tasks:

1. Revisit the case of the Lorenz–1960 model, this time using a hard-constrained PINN. Is your result more accurate than the soft-constrained PINN?
2. Revisit the harmonic oscillator in both the first-order and second-order representation using hard constraints. For the second-order representation, note that you will have to construct a solution ansatz such that $u^{\boldsymbol{\theta}}(0) = u_0$ and $\frac{\mathrm{d}u^{\boldsymbol{\theta}}}{\mathrm{d}t}(0) = u_0'$. To construct such a solution ansatz, you can use a second-order Taylor series approximation.
3. The hard constraint (2.3) is only one possible way to enforce that the neural network solution ansatz $u^{\boldsymbol{\theta}}(t)$ satisfies the initial condition. One obvious downside of this ansatz is that for longer time intervals, the multiplying factor t grows and as such the neural network $\mathcal{N}^{\boldsymbol{\theta}}(t)$ will have to offset this growth. There are other ways how hard constraints can be constructed which can prevent this linear growth. Come up with at least two different hard constraints, and test them out for the harmonic oscillator represented as a system of two equations.

2.5 Why this project is relevant

Scientific machine learning, and the subfield of physics-informed machine learning have seen an exponential increase of interest in the last 5 years only. Today, almost all fields of science use machine learning as a main tool to advance knowledge, which is an unprecedented scientific revolution. (Scientific) machine learning is at the heart of the 2024 Nobel prices in physics and chemistry, respectively. Scientific machine learning has led to breakthroughs in areas such as protein folding [4] and weather prediction [1]. As such, getting familiar with scientific machine learning is of critical importance if you plan on pursuing research or a career in the mathematical sciences.

2.6 Further reading

Since the field of scientific machine learning is quickly evolving, so is the associated literature, and this necessitates keeping up with the current literature in the field. For one relatively recent review paper on this subject, see [2].

Bibliography

[1] Bi K., Xie L., Zhang H., Chen X., Gu X. and Tian Q., Accurate medium-range global weather forecasting with 3D neural networks, *Nature* **619** (2023), 533–538.

[2] Cuomo S., Di Cola V.S., Giampaolo F., Rozza G., Raissi M. and Piccialli F., Scientific machine learning through physics–informed neural networks: where we are and what's next, *J. Sci. Comput.* **92** (2022), 88.

[3] Goodfellow I., Bengio Y. and Courville A., *Deep learning*, MIT press, 2016.

[4] Jumper J., Evans R., Pritzel A., Green T., Figurnov M., Ronneberger O., Tunyasuvunakool K., Bates R., Žídek A., Potapenko A. *et al.*, Highly accurate protein structure prediction with AlphaFold, *nature* **596** (2021), 583–589.

[5] Krantz S.G., *A primer of mathematical writing*, vol. 243, American Mathematical Soc., 2017.

[6] Lagaris I.E., Likas A. and Fotiadis D.I., Artificial neural networks for solving ordinary and partial differential equations, *IEEE Trans. Neural Netw.* **9** (1998), 987–1000.

[7] Liu Z., Wang Y., Vaidya S., Ruehle F., Halverson J., Soljačić M., Hou T.Y. and Tegmark M., KAN: Kolmogorov-Arnold networks, *arXiv preprint arXiv:2404.19756* (2024).

[8] Lorenz E.N., Maximum simplification of the dynamic equations, *Tellus* **12** (1960), 243–254.

[9] Raissi M., Perdikaris P. and Karniadakis G.E., Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, *J. Comput. Phys.* **378** (2019), 686–707.

[10] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A.N., Kaiser L. and Polosukhin I., Attention is all you need, in *Advances in Neural Information Processing Systems*, vol. 30, edited by I. Guyon, U.V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan and R. Garnett, Curran Associates, Inc., vol. 30, 2017 .