Neural Ordinary Differential Equations

David Tran and Spencer Kelly

March 25, 2024

Abstract

In this final lab of a series of 4 labs exploring numerical methods, we discuss numerical ODE solvers. We describe Euler's method and the family of Runge-Kutta methods and the computational tradeoffs between them. In particular, we explore its usage in neural ordinary differential equations, and how different families of ODE solvers provide different tradeoffs on the approximation accuracy of the model and its computational cost.

1 Introduction

Neural ODEs are set up such that the hidden state at any time t is such that it returns the function f for the differential equation $\frac{dh(t)}{dt} = f(h(t), t, \theta)$. That is, for some system with state $h(t)$, we can train the neural network to find $\frac{dh(t)}{dt}$ for any time t. This means that we are essentially training the ODE to act as a b̈lack-boxs̈olver of some dynamical system in which the state of our system can be described as $h(t)$ so that the rate of change of our system can be described as $\frac{dh(t)}{dt} = f(h(t), t, \theta)$. But to teach the model to act as the differential equation and use it as a black-box solver for our system, we must have an underlying method for solving such a system. That's to say, all the model can do is to try its best to act as the underlying differential equation for the system, but in order to solve the system, we need a way to integrate the differential equation.

There are various numerical integration methods with varying levels of accuracy and computational intensity. In this lab, we will go over some effective numerical methods for integration and their usefulness. We then show how to these numerical methods can be used in a neural ODE to provide solutions to systems that don't have an obvious differential equation that governs them. We evaluate the difference in effectiveness of these numerical methods when used in a neural ODE, and touch the on broader applications of neural ODEs.

2 Differential Equation Solvers

In this section, we discuss two different methods of numerical differential equation solving, their accuracy, and their importance.

2.1 Euler's Method

There are a variety of methods that can be used for numerical integration, these range all the way from methods such as using random walks in $\mathbb{Z}^d$ to solve for harmonic functions, to the simple and straightforward Euler's method on $\mathbb{R}$. Of these methods there are two that stand-out as some of the most effective, and will be used in this lab. The aforementioned Euler's method is a simple yet

effective way to solve a differential equation, and will be one of the two examined in this lab. The other is from the Runge-Kutta family, which are more computationally expensive but often give a more accurate solution.

Euler's method is rather simple, given some step size s, some initial value $h(t_0)$ and the differential equation $\frac{dh(t)}{dt}$, it moves along the tangent vector in the direction of $\frac{dh(t)}{dt}$ for some distance that has a projected length of s along the t axis. The equation to represent this looks as such, for some $t \in (t_0, t_1)$, $h_{n+1} = h_n + s\frac{dh(t)}{dt}$. This equation is evaluated for each step $t_n = n \cdot s \in (t_0, t_1)$, where $n \in \mathbb{N}$ is the step number. This is an effective way of integrating a differential equation, though it is subject to errors that can propagate as the step size increases.

Assuming at time t you lie on the one point to another, you lose accuracy by approximating the curve to be the straight line in the direction of $\frac{dh(t_i)}{dt}$ for time s. This introduces an error corresponding to the difference between the point on the line $h(t_i + s)$ and the point at the end of straight line from $(t_i, h(t_i))$ to $(t_i + s, h(t_i) + s\frac{dh(t_i)}{dt})$. Not only this, but you have to keep in mind that the latter point is not necessarily on the solution curve, so that the $\frac{dh(t)}{dt}$ vector field may look very different at the latter point than at the former point, and this introduces yet another source of error. That is, as you encounter per-step errors by approximating the curve to be a straight line, you also encounter a global error resulting from falling off the original solution curve for the IVP and onto what would be other integral curves for the vector field $\frac{dh(t)}{dt}$.
Euler's method offers simplicity at the price of such errors.

2.2 Runge-Kutta Methods

As for the Runge-Kutta family, they are better designed to avoid errors, at the cost of increased computational cost. As implied, these methods are more involved than that of Euler's, giving a system of equations at each step in the integration, as opposed to a single equation. The group of Runge-Kutta methods used in this lab, arguably a favourite for computational initial value problem-solving functions, fall into those of the 4th and 5th order. This means that the Runge-Kutta methods in this lab serve to calculate each time steps accuracy to that of the 4th and 5th order terms.

The order of a Runge-Kutta method is the number of terms it uses when calculating the slope of the vector to follow on the next step. For example, a Runge-Kutta method of the 4th order does the following at step i of the iterative method for some step size s, and differential equation $\frac{dh(t)}{dt} = f(t, h)$:

$$h_{n+1} = h_n + \frac{s}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$
$$t_{i+1} = t_i + s$$

Where

$$k_1 = f(t_i, h_i)$$
$$k_2 = f(t_i + \frac{s}{2}, h_i + \frac{k_1}{2})$$
$$k_3 = f(t_i + \frac{s}{2}, h_i + \frac{k_2}{2})$$
$$k_4 = f(t_i + s, h_i + k_3)$$

Here we can see that within any given step of this method, the differential equation is evaluated at the start point of the interval $[t_i, t_i + s]$, giving us k_1 Once we have k_1, we use that slope to calculate the estimated slope at the middle point of the interval, which gives us k_2. With k_2, we

again calculate the slope at the midpoint of the interval but this time using the slope k_2, which gives us k_3. Finally, we calculate the slope at the endpoint of the interval, using k_3, and giving us k_4. Once we have these, we use a weighted sum of them to calculate the next point on the integral curve. It is important to note that this example is used to display the idea behind Runge-Kutta methods, and that the multi-stage slope evaluation and subsequent weighted averaging within each step is used across the entire Runge-Kutta family.

As is evident from above, the Runge-Kutta 4th order method is a much more complex method than Euler's method, and the Runge-Kutta method used in this lab called dopri5 (The Dormand-Prince Method), is even more sophisticated than the classic Runge-Kutta 4th order method. The Dormand-Prince method uses the same Runge-Kutta method as displayed in the 4th order, but does so also in the 5th order, along with adapting the step size for each individual step based on error calculations obtained by contrasting the 4th and 5th order coefficients (as opposed to using a fixed step size, seen in Euler's method and the Runge-Kutta 4th order method). dorpi5 is able to balance error-mitigation with computational efficiency to provide accurate solutions in the 5th order without sacrificing too much computational cost, which is why it is the default of ODE solver functions in MATLAB and other computational software.

3 Neural Ordinary Differential Equations

Neural Ordinary Differential Equations (Neural ODEs) is a deep learning approch that uses neural networks to model continuous time-series data. Unlike traditional deep learning models that rely on discrete layers and fixed architectures, Neural ODEs leverage the theory of ordinary differential equations (ODEs) to describe dynamic systems. In essence, they encapsulate the evolution of hidden states continuously over time, offering a flexible framework for modeling complex temporal dynamics.

Existing models for time-series data such as residual networks and recurrent neural networks vompose sequences of transformation to some hidden state h as $h_{t+1} = h_t + f(h_t, \theta_t)$. Crucially, these models use a fixed number of layers to represent discrete (pre-determined) time steps $t_1, \ldots, t_n$. The neural ODE instead considers the sequence of transformations as a continuous and represents the transformation as an ordinary diferential equation: $\frac{dh}{dt} = f(h(t), t, \theta)$. The neural network is then trained to approximate the function f as $\hat{f}$. The output of the model $h(t)$ at some time t is computed from an initial value $h(0)$ (the input data) using any ODE solver on the ODE

$$\frac{dh(t)}{dt} = \hat{f}(h(t), t, \theta)$$

3.1 Why machine learning for DE solving

The advantage of machine learning for DE solving over traditional analytical or other numerical approximation methods is due to their flexibility in approximating relations of arbitrary complexity. Due to the performance of the model being a function of the amount of data available on the relation-of-interest, neural networks are particularly advantageous for solving differential equations for which the knowledge of the underlying dynamics of the relation are unknown or limited, compared to the large amount of data representing the relation.

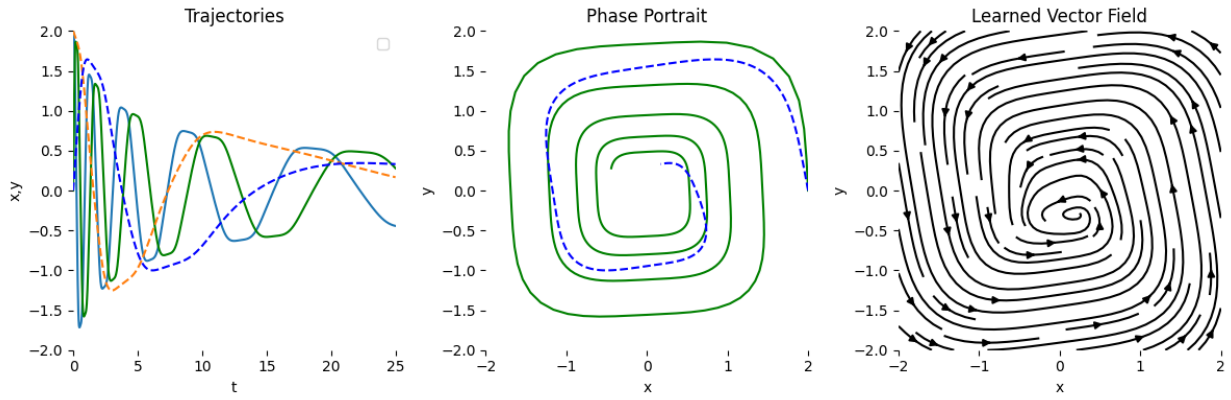

Figure 1: The predicted trajectory, and the corresponding learned vector field after one iteration. The green represents the ground truth, while the blue represents the output of the model. The Dormand-Prince method, a type of Runge-Kutta ODE solver, is used to evaluate the final prediction.

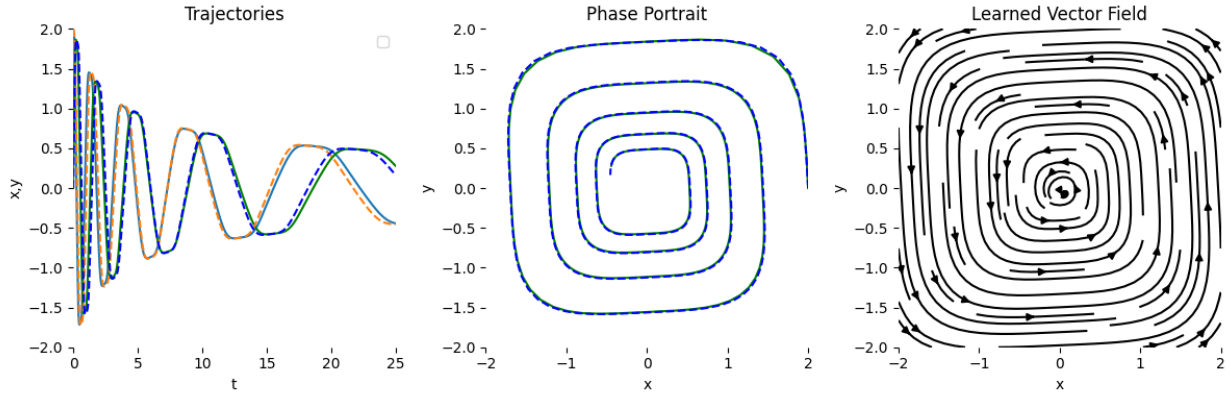

Figure 2: After 2000 iterations, using Dormand-Prince.

3.2 Application

We use the implementation of the neural ODE described in [**?**] using the code in [**?**]. We use it to learn the dynamics of a simple harmonic oscillator with slight dampening. Observe in Figure **??** how the predicted trajectories and phase portrait (blue) do not match very well the ground truth (green). Although the shape of the learned vector field looks accurate, it is askew from the proper orientation of the phase portrait In Figure **??**, the predicted trajectory and phase portrait nearly perfectly coincide, and we observe that the learned vector field nearly matches what one would expect from the phase portrait.

3.2.1 Euler's Method vs Runge-Kutta

In the above section, after learning the function $\frac{dh(t)}{dt}$, the model uses the Dormand-Prince method, an adaptive-step ODE solver that falls under the family of Runge-Kutta methods. We now compare its performance to using a fixed-point ODE solver, namely, Euler's method.

As seen in Figure **??**, after the first iteration, the trajectory and phase portrait match the

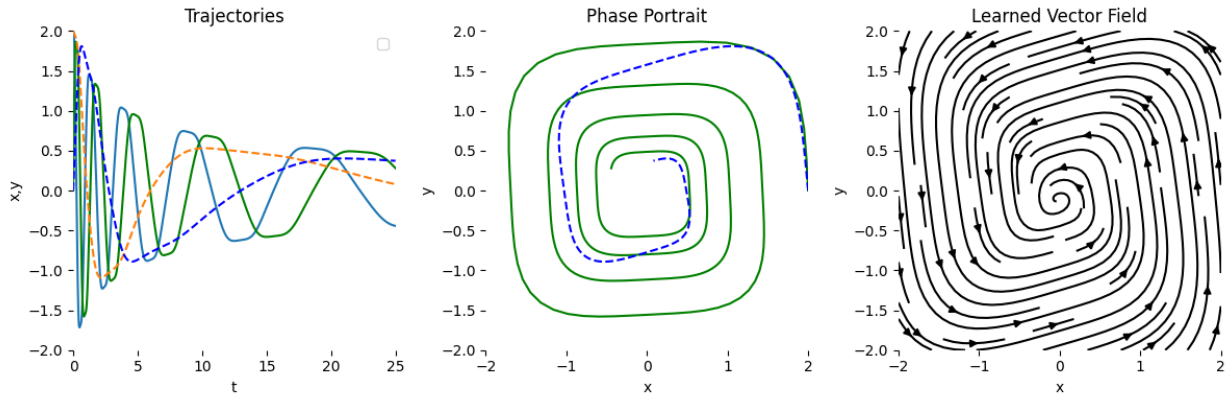

Figure 3: After one iteration, using Euler's method.

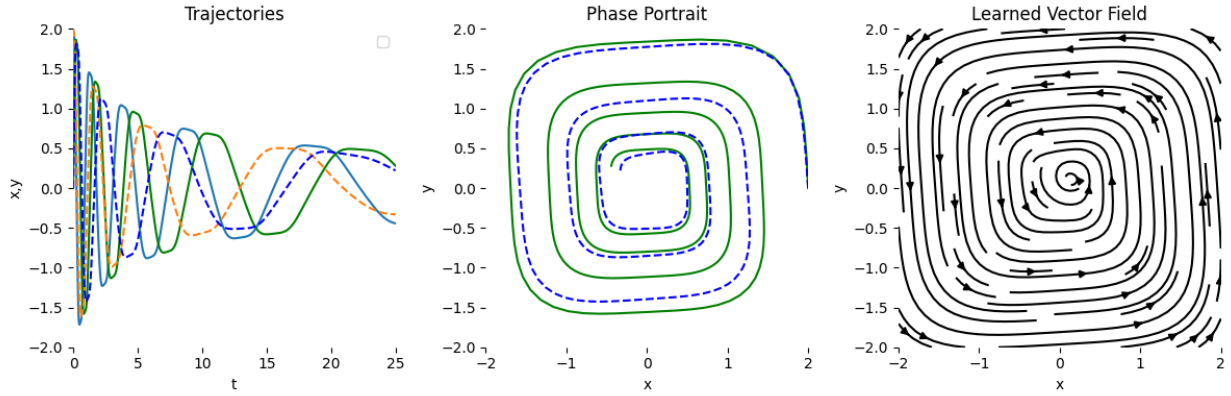

Figure 4: After 2000 iterations, using Euler's method.

ground-truth much worse, and the learned vector field exhibits greater distortion than when using the Dormand-Prince method as the ODE solver. Even after 2000 iterations as in Figure **??**, we see the model is not able to capture the trajectory as well as when using the Dormand-Prince method, as in Figure **??**.

When using the Dormand-Prince method, training and evaluating the model took approximately 4 minutes and 53 seconds on an Apple M1 Pro, consisting of 8 physical cores each with 8 logical processors. Using Euler's method, training and evaluating the model took approximately 2 minutes and 3 seconds, less than half the time of the Dormand-Prince method. This demonstrates the cost-accuracy tradeoff between the two methods, and more generally, between fixed-step and adaptive-step ODE solvers.

4 Conclusion

We discussed the difference between fixed-step and adaptive-step numerical ODE solvers, in particular, Euler's method and Runge-Kutta methods. We described neural ODEs, a deep learning approach that chooses to model transformations to the input data as an ordinary differential equation, allowing one to leverage the rich theory of ODE solvers to the problem of time-series modelling. We showed that, when using the Runge-Kutta methods in neural ODEs, we observe better approx-

imations at the cost of longer compute time, and vice-versa for Euler's method.

The team was able to collaborate well and properly divide the tasks. We were able to leverage our existing experience in ODE solvers in applications for simulating fluid dynamics, as well as experience in novel machine learning research. In the future, it would be interesting to apply neural ODEs to other time-series problems with complex system dynamics, such as fluid flow simulation, weather prediction, and financial modelling.

5 Appendix

```
SOLVERS = {
  'dopri8': Dopri8Solver,
  'dopri5': Dopri5Solver,
  'bosh3': Bosh3Solver,
  'fehlberg2': Fehlberg2,
  'adaptive_heun': AdaptiveHeunSolver,
  'euler': Euler,
  'midpoint': Midpoint,
  'heun3': Heun3,
  'rk4': RK4,
  'explicit_adams': AdamsBashforth,
  'implicit_adams': AdamsBashforthMoulton,
  # Backward compatibility: use the same name as before
  'fixed_adams': AdamsBashforthMoulton,
  # ~Backwards compatibility
  'scipy_solver': ScipyWrapperODESolver,
}


def odeint(func, y0, t, *, rtol=1e-7, atol=1e-9, method=None, options=None,
    event_fn=None):

  shapes, func, y0, t, rtol, atol, method, options, event_fn, t_is_reversed =
    _check_inputs(func, y0, t, rtol, atol, method, options, event_fn, SOLVERS)

  solver = SOLVERS[method](func=func, y0=y0, rtol=rtol, atol=atol, **options)

  if event_fn is None:
      solution = solver.integrate(t)
  else:
      event_t, solution = solver.integrate_until_event(t[0], event_fn)
      event_t = event_t.to(t)
      if t_is_reversed:
          event_t = -event_t

  if shapes is not None:
      solution = _flat_to_shape(solution, (len(t),), shapes)

  if event_fn is None:
      return solution
  else:
      return event_t, solution
```

Listing 1: Code for the ODE solver.

```
  class ODEFunc(nn.Module):

```

```
    def __init__(self):
        super(ODEFunc, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(2, 50),
            nn.Tanh(),
            nn.Linear(50, 2),
        )

        for m in self.net.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=0.1)
                nn.init.constant_(m.bias, val=0)

    def forward(self, t, y):
        return self.net(y**3)


class RunningAverageMeter(object):
    """Computes and stores the average and current value"""

    def __init__(self, momentum=0.99):
        self.momentum = momentum
        self.reset()

    def reset(self):
        self.val = None
        self.avg = 0

    def update(self, val):
        if self.val is None:
            self.avg = val
        else:
            self.avg = self.avg * self.momentum + val * (1 - self.momentum)
        self.val = val


if __name__ == '__main__':

    ii = 0

    func = ODEFunc().to(device)

    optimizer = optim.RMSprop(func.parameters(), lr=1e-3)
    end = time.time()

    time_meter = RunningAverageMeter(0.97)

    loss_meter = RunningAverageMeter(0.97)

    for itr in range(1, args.niters + 1):
        optimizer.zero_grad()
        batch_y0, batch_t, batch_y = get_batch()
        pred_y = odeint(func, batch_y0, batch_t, method='euler').to(device)
        loss = torch.mean(torch.abs(pred_y - batch_y))
        loss.backward()
        optimizer.step()

        time_meter.update(time.time() - end)
```

```
        loss_meter.update(loss.item())

        if itr % args.test_freq == 0:
            with torch.no_grad():
                pred_y = odeint(func, true_y0, t)
                loss = torch.mean(torch.abs(pred_y - true_y))
                print('Iter {:04d} | Total Loss {:.6f}'.format(itr, loss.item()))
                visualize(true_y, pred_y, func, ii)
                ii += 1

        end = time.time()
```

Listing 2: Code for training the network on the oscillator.

References

[1] Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018). *Neural Ordinary Differential Equations*. Advances in Neural Information Processing Systems.

[2] Chen, R. T. Q. (2018). *torchdiffeq*. Retrieved from `https://github.com/rtqichen/torchdiffeq`