

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

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

2.2 Runge-Kutta Methods

3 Neural Ordinary Differential Equations

Neural Ordinary Differential Equations (Neural ODEs) is a deep learning approach 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 vcompose 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, \dots, t_n . The neural ODE instead considers the sequence of transformations as a continuous and represents the transformation as an ordinary differential 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

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

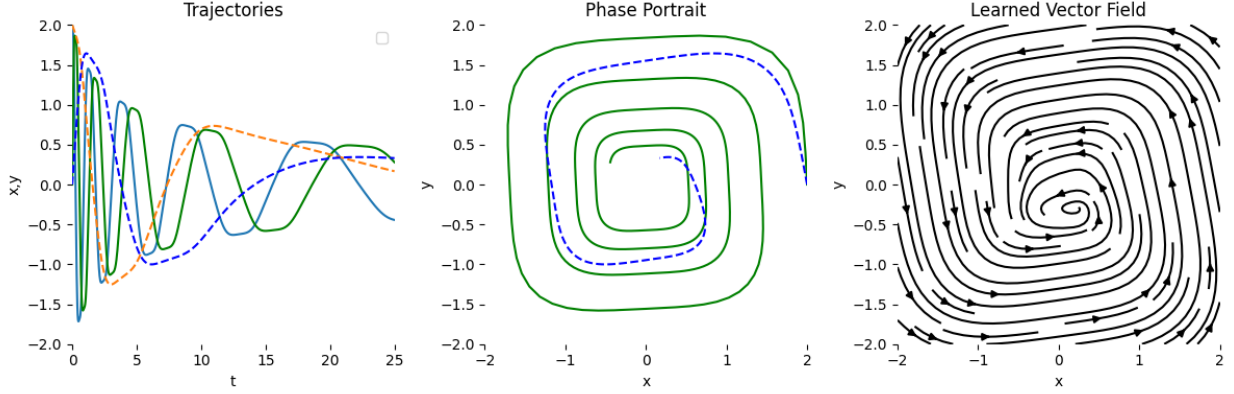


Figure 1: The predicted trajectory, and the corresponding learned vector field after one iteration. The green represents the ground truth, while the blue represents the output of the model. The Dormand-Prince method, a type of Runge-Kutta ODE solver, is used to evaluate the final prediction.

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.

3.2 Application

We use the implementation of the neural ODE described in [1] using the code in [2]. We use it to learn the dynamics of a simple harmonic oscillator with slight dampening. Observe in Figure 1 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 2, 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 3, after the first iteration, the trajectory and phase portrait match the 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 4, we see the model is not able to capture the trajectory as well as when using the Dormand-Prince method, as in Figure 2.

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

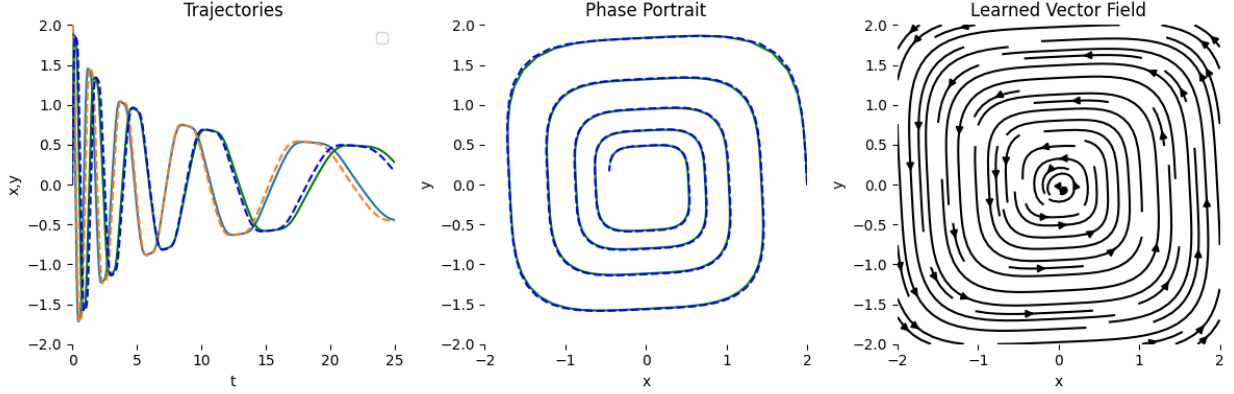


Figure 2: After 2000 iterations, using Dormand-Prince.

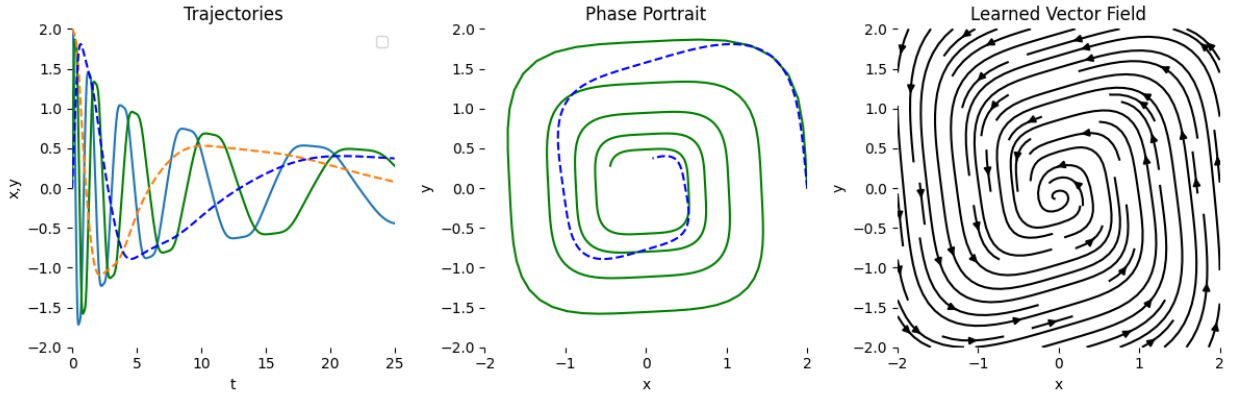


Figure 3: After one iteration, using Euler's method.

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 approximations 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.

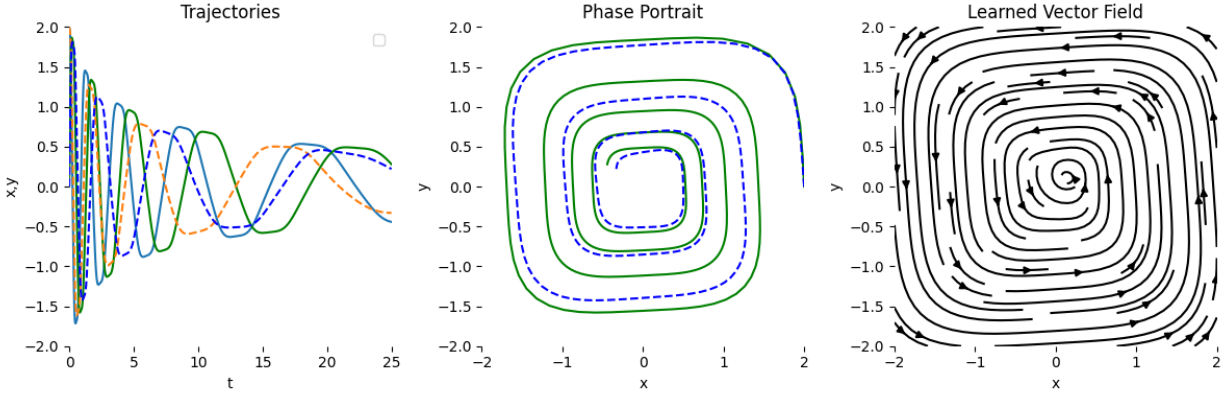


Figure 4: After 2000 iterations, using Euler's method.

5 Appendix

```

1 SOLVERS = {
2     'dopri8': Dopri8Solver,
3     'dopri5': Dopri5Solver,
4     'bosh3': Bosh3Solver,
5     'fehlberg2': Fehlberg2,
6     'adaptive_heun': AdaptiveHeunSolver,
7     'euler': Euler,
8     'midpoint': Midpoint,
9     'heun3': Heun3,
10    'rk4': RK4,
11    'explicit_adams': AdamsBashforth,
12    'implicit_adams': AdamsBashforthMoulton,
13    # Backward compatibility: use the same name as before
14    'fixed_adams': AdamsBashforthMoulton,
15    # ~Backwards compatibility
16    'scipy_solver': ScipyWrapperODESolver,
17 }
18
19
20 def odeint(func, y0, t, *, rtol=1e-7, atol=1e-9, method=None, options=None,
21           event_fn=None):
22
23     shapes, func, y0, t, rtol, atol, method, options, event_fn, t_is_reversed =
24         _check_inputs(func, y0, t, rtol, atol, method, options, event_fn, SOLVERS)
25
26     solver = SOLVERS[method](func=func, y0=y0, rtol=rtol, atol=atol, **options)
27
28     if event_fn is None:
29         solution = solver.integrate(t)
30     else:
31         event_t, solution = solver.integrate_until_event(t[0], event_fn)
32         event_t = event_t.to(t)
33         if t_is_reversed:
34             event_t = -event_t
35
36     if shapes is not None:
37         solution = _flat_to_shape(solution, (len(t),), shapes)

```

```

37 if event_fn is None:
38     return solution
39 else:
40     return event_t, solution

```

Listing 1: Code for the ODE solver.

```

1 class ODEFunc(nn.Module):
2
3     def __init__(self):
4         super(ODEFunc, self).__init__()
5
6         self.net = nn.Sequential(
7             nn.Linear(2, 50),
8             nn.Tanh(),
9             nn.Linear(50, 2),
10        )
11
12        for m in self.net.modules():
13            if isinstance(m, nn.Linear):
14                nn.init.normal_(m.weight, mean=0, std=0.1)
15                nn.init.constant_(m.bias, val=0)
16
17        def forward(self, t, y):
18            return self.net(y**3)
19
20
21 class RunningAverageMeter(object):
22     """Computes and stores the average and current value"""
23
24     def __init__(self, momentum=0.99):
25         self.momentum = momentum
26         self.reset()
27
28     def reset(self):
29         self.val = None
30         self.avg = 0
31
32     def update(self, val):
33         if self.val is None:
34             self.avg = val
35         else:
36             self.avg = self.avg * self.momentum + val * (1 - self.momentum)
37         self.val = val
38
39
40 if __name__ == '__main__':
41
42     ii = 0
43
44     func = ODEFunc().to(device)
45
46     optimizer = optim.RMSprop(func.parameters(), lr=1e-3)
47     end = time.time()
48
49     time_meter = RunningAverageMeter(0.97)
50
51     loss_meter = RunningAverageMeter(0.97)
52

```

```

53     for itr in range(1, args.niters + 1):
54         optimizer.zero_grad()
55         batch_y0, batch_t, batch_y = get_batch()
56         pred_y = odeint(func, batch_y0, batch_t, method='euler').to(device)
57         loss = torch.mean(torch.abs(pred_y - batch_y))
58         loss.backward()
59         optimizer.step()
60
61         time_meter.update(time.time() - end)
62         loss_meter.update(loss.item())
63
64         if itr % args.test_freq == 0:
65             with torch.no_grad():
66                 pred_y = odeint(func, true_y0, t)
67                 loss = torch.mean(torch.abs(pred_y - true_y))
68                 print('Iter {:04d} | Total Loss {:.6f}'.format(itr, loss.item()))
69                 visualize(true_y, pred_y, func, ii)
70                 ii += 1
71
72     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>