Continuous Time Integration in Neuromancer

May 13, 2022

Introduction

Our recent development work in Neuromancer has given us the capability to learn dynamical systems of the form:

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t)) \tag{1}$$

or

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t), t) \tag{2}$$

or

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t). \tag{3}$$

where $\mathbf{x}(t)$ is the time-varying state of the considered system, $\mathbf{u}(t)$ are system control inputs, and $\mathbf{f}$ is the state transition dynamics. This modeling strategy can be thought of as an equivalent method to *Neural Ordinary Differential Equations* [1], whereby an ODE of the above forms is fit to data with a universal function approximator (e.g. deep neural network) acting as the state transition dynamics. To train an appropriate RHS, Chen et al. utilize a continuous form of the adjoint equation; itself solved with an ODESolver. Instead, we choose to utilize the autodifferentiation properties of PyTorch to build differentiable canonical ODE integrators (e.g. as in Raissi et al. [2]).

We wish to test the capability of this methodology in a variety of situations and configurations. Of particular interest is the predictive capability of this class of methods compared with Neural State Space Models and other traditional "black-box" modeling techniques.

Before moving on, it is important to note that there are two dominant neural ODE packages freely available. The first is *DiffEqFlux.jl* developed and maintained by SciML within the Julia ecosystem. The second is *torchdyn* which lives within the PyTorch ecosystem. Both packages are well-documented and have become established in application-based research literature.

1 Syntax and Usage

Two Neuromancer dynamics classes handle continuous time dynamics: `ODEAuto` and `ODENonAuto`. As their names suggest, these classes handle the scenarios corresponding to Equations 1-2. Their usage is detailed below:

1.1 Autonomous ODEs

Autonomous ODEs are those that do not explicitly depend on time; as such, the dynamics are functions of state variables alone. A fully-specified neural ODE of this type requires a RHS function (either a neural network or other tensor-tensor mapping. The use of the `ODEAuto` class is as follows:

```python
# Instantiate the ODE RHS as MLP:
fx = blocks.MLP(nx, nx, linear_map=slim.maps['linear'],
                nonlin=activations['leakyrelu'],
                hsizes=[10, 10])

# Instansiate the integrator, handing it the RHS "fx":
fxRK4 = integrators.RK4(fx, h=ts)

# Identity output mapping:
fy = slim.maps['identity'](nx, nx)

# Creating the dynamics model:
NODE = dynamics.ODEAuto(fxRK4, fy, name='dynamics',
                        input_key_map={"x0": f"x0_{estim.name}"})
```

Note that the transition dynamics `fx` is a square mapping ($nx \to nx$) of states as expected.

1.2 Non-Autonomous ODEs

Non-autonomous ODEs depend on time, external inputs, or both time and inputs. The syntax for these systems changes as continuos-time representations of time and any external inputs must be available and provided to the integrator. This is handled with the construction and passing of interpolants.

1.2.1 Time as input

An example non-autonomous system with explicit dependence on time is the *Forced Duffing Oscillator*, given by the ODEs:

$$\frac{dx_1}{dt} = x_2 \tag{4}$$

$$\frac{dx_2}{dt} = -\delta x_2 - \alpha x_1 - \beta x_1^3 + \gamma \cos(\omega t) \tag{5}$$

Supposing that the model is known except for one or more of the parameters, one can build a consistent tensor-tensor mapping to pass to an integrator and `ODENonAuto` class. First, the ODE RHS is defined:

```python
class DuffingParam(ODESystem):
    def __init__(self, insize=3, outsize=2):
        """
        :param insize:
        :param outsize:
        """
        super().__init__(insize=insize, outsize=outsize)
        self.alpha = nn.Parameter(torch.tensor([1.0]), requires_grad=False)
        self.beta = nn.Parameter(torch.tensor([5.0]), requires_grad=False)
        self.delta = nn.Parameter(torch.tensor([0.02]), requires_grad=False)
        self.gamma = nn.Parameter(torch.tensor([8.0]), requires_grad=False)
        self.omega = nn.Parameter(torch.tensor([0.5]), requires_grad=True)

    def ode_equations(self, x):
        # states
        x0 = x[:, [0]]  # (# batches,1)
        x1 = x[:, [1]]
        t =  x[:, [2]]
        # equations
        dx0dt = x1
```

```
        dx1dt = -self.delta*x1 - self.alpha*x0 - self.beta*x0**3 +
                self.gamma*torch.cos(self.omega*t)
        return torch.cat([dx0dt, dx1dt], dim=-1)
```

Note that in this definition, only the paramter ω is tunable; thus, this is a 1-parameter training task. Additionally, note the dimensionality: expected is a state dimension of three, with the third dimension corresponding to time. The specification of the Neural ODE begins with defining a continuous representation of time:

```
t = torch.from_numpy(t)
interp_u = LinInterp_Offline(t, t)
```

The rest of the setup is identical to the autonomous case with the exception of the dynamics class:

```
# Instantiate the ODE RHS:
duffing_sys = ode.DuffingParam()

# Instansiate the integrator, handing it the RHS "duffing_sys":
fxRK4 = integrators.RK4(duffing_sys, interp_u=interp_u, h=ts)

# Identity output mapping:
fy = slim.maps['identity'](nx, nx)

# Creating the dynamics model:
dynamics_model = dynamics.ODENonAuto(fxRK4, fy,
    input_key_map={"x0": f"x0_{estim.name}", "Time": "Timef", 'Yf': 'Yf'},
    name='dynamics',     # must be named 'dynamics' due to some issue in visuals.py
    online_flag=False
    )
```

1.2.2 Other external inputs

Control signals are dealt with in the same manner as time: they must first be represented in a continuous form via an interpolant. Specification of these interpolants is as follows:

```
t = torch.from_numpy(t) # from numpy dataset
u = raw['U'].astype(np.float32) # getting control 'u' from data dictionary
u = np.append(u,u[-1,:]).reshape(-1,2)
ut = torch.from_numpy(u)
interp_u = LinInterp_Offline(t, ut)
```

The neural ODE is specified in the same way as the forced Duffing system:

```
# Get the dimension of extra inputs
nu = dims['U'][1]

# Construct black-box RHS mapping from nx+nu to nx
black_box_ode = blocks.MLP(insize=nx+nu, outsize=nx, hsizes=[30, 30],
                           linear_map=slim.maps['linear'],
                           nonlin=activations['gelu'])

# Hand it over to the integrator with the interpolant:
fx_int = integrators.RK4(black_box_ode, interp_u=interp_u, h=modelSystem.ts)

# Identity output mapping:
fy = slim.maps['identity'](nx, nx)
```

```python
# Creating the dynamics model:
dynamics_model = dynamics.ODENonAuto(fx_int, fy, extra_inputs=['Uf'],
                input_key_map={"x0": f"x0_{estim.name}", "Time": "Timef", 'Yf': 'Yf'},
                name='dynamics',
                online_flag=False
                )
```

References

[1] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[2] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.