A. Code

Built-in to `torchdiffeq` `torchdiffeq` (Chen et al., 2018) has chosen to support adjoint seminorms as a built-in option. This may be used by passing `odeint_adjoint(..., adjoint_options=dict(norm='seminorm'))`.

PyTorch reference implementation A reference implementation, using PyTorch (Paszke et al., 2019) and `torchdiffeq` (Chen et al., 2018), is shown in Figure 7.

By default, `torchdiffeq` uses a (somewhat unusual) mixed L^∞-RMS norm. Given the adjoint state $[a_t, z, a_z, a_\theta]$, then the norm used is

$$\begin{aligned}\|[a_t, z, a_z, a_\theta]\| = \max\{ &\|a_t\|_{\mathrm{RMS}}, \\ &\|z\|_{\mathrm{RMS}}, \\ &\|a_z\|_{\mathrm{RMS}}, \\ &\|a_\theta\|_{\mathrm{RMS}}\},\end{aligned}$$

where for $x = (x_1, \ldots, x_n)$,

$$\|x\|_{\mathrm{RMS}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}.$$

Respecting this convention, the code provided reduces this to the seminorm

$$\|[a_t, z, a_z, a_\theta]\| = \max\{\|z\|_{\mathrm{RMS}}, \|a_z\|_{\mathrm{RMS}}\}.$$

Other similar approaches such as pure-RMS over $[z, a_z]$ should be admissible as well.

Julia The DifferentialEquation.jl library (Rackauckas & Nie, 2017) library offers an argument `internalnorm` which may be used to implement seminorms in a manner analogous to the reference implementation provided here.

B. Experimental details

B.1. Neural Controlled Differential Equations

We use the same setup and hyperparameters as in Kidger et al. (2020). The loss function is cross entropy. The optimiser used was Adam (Kingma & Ba, 2015), with learning rate 1.6×10^{-3}, batch size of 1024, and 0.01-weighted L^2 weight regularisation, trained for 200 epochs. The number of hidden channels (the size of z) is 90, and f is parameterised as a feedforward network, of width 40 with 4 hidden layers, ReLU activation, and tanh final activation.

B.2. Continuous Normalising Flows

We follow Grathwohl et al. (2019) and Finlay et al. (2020). The loss function is the negative log likelihood $-\log(p(z(T) = x))$ of equation (6). The optimiser used was Adam, with learning rate 10^{-3} and batch size 256, trained for 100 epochs. Relative and absolute tolerance of the solver are both taken to be 10^{-5}. We used a multi-scale architecture as in Grathwohl et al. (2019) and Finlay et al. (2020), with 4 blocks of CNFs at 3 different scales.

B.3. Symplectic ODE-Net

The optimiser used was Adam, with learning rate 10^{-3}, batch size of 256, and 0.01-weighted L^2 weight regularisation. Relative and absolute tolerance of the solver are both taken to be 10^{-4}. We use the same architecture as in Zhong et al. (2020): this involves parameterising H as a sum of kinetic and potential energy terms. These details of Sympletic ODE-Net are a little involved, and we refer the reader to Zhong et al. (2020) for full details.

```python
import torchdiffeq

def rms_norm(tensor):
    return tensor.pow(2).mean().sqrt()

def make_norm(state):
    state_size = state.numel()

    def norm(aug_state):
        y = aug_state[1:1 + state_size]
        adj_y = aug_state[1 + state_size:1 + 2 * state_size]
        return max(rms_norm(y), rms_norm(adj_y))
    return norm

torchdiffeq.odeint_adjoint(func=..., y0=..., t=...,
                           adjoint_options=dict(norm=make_norm(y0)))
```

Figure 7: Reference PyTorch implementation for adjoint seminorms.