CS-598: Deep Generative and Dynamical Models **Spring 2023**

Topic 24: Neural Ordinary Differential Equations 1

Lecturer: Arindam Banerjee *Scribe: Nicolas Nytko, Sean Farhat, Nathanael Assefa*

24.1 Neural Ordinary Differential Equations

24.1.1 Ordinary Differential Equations

A simple ordinary differential equation has the following form, modelling the dynamics of a system:

$$\frac{dz(t)}{dt} = f(z(t), t, \theta) \tag{24.1}$$

Given an initial state $z(0)$, and we can ask an ODE solver to tell us the value at a desired time $z(T)$, if we provide the dynamics above.

ODE solvers in general are numerically based. That is, they will discretize the path in time: $0, s, 2s, \ldots, T$, where s is the step size, and evaluate the path iteratively:

$$z(t+1) = z(t) + sf(z(t), t, \theta) \tag{24.2}$$

This is known as the **forward Euler method** (Figure 24.1), and can been seen as an approximation of the dynamics of Equation 24.1, where instead of the limit that defines the derivative, we use the fixed step size s. The smaller the step size, the better the accuracy of the solver.

More complex methods exist that take advantage of higher order information, such as the Runge-Kutta or Backward differentiation families of numerical integrators, though their implementation is out of the scope of this note.

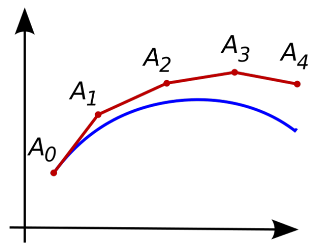


FIGURE 24.1: The forward Euler method

24.1.2 Motivating Neural ODE's

Now, let's look at a familiar building block of neural networks: the residual layer (Figure 24.2)

The idea here is simple: the output at a particular layer should include information about the input and its transformed version. For a layer f at layer t parameterized by θ_t,

$$z_{t+1} = z_t + f(z_t, \theta_t) \tag{24.3}$$

This looks familiar, doesn't it? It is exactly the forward Euler method that an ODE solver uses, i.e. Equation 24.1 with $s = 1$.

So, we can view a specifically designed residual neural network as an ODE solver, where

1. The input is the initial condition $z(0)$
2. The output is the system at a desired timestep $z(T)$
3. Each layer does one step of the Euler discretization
4. The number of layers determines how well our approximation comes close to the true continuous dynamics of the system

5. The neural network itself models the dynamics

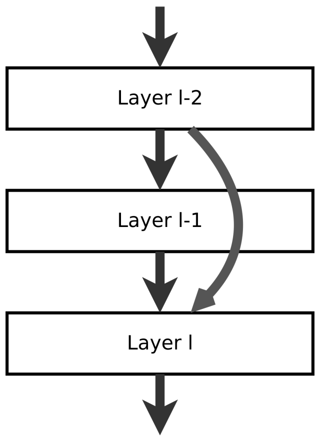


FIGURE 24.2: A series of residual layers

What's even nicer is that with this formulation, we can probe the network at a particular layer L and get the solution to the dynamics at that time $z(t_L)$.

24.1.3 Learning

In the same vein as supervised learning where we have a data pair (x, y), we will have data points where $(x, y) = (z(t_0), z(t_1))$, that is the initial and ending positions of our function. We can think of t as a time variable, so we wish to learn the dynamics for how z changes from time t_0 to t_1.

How do we compute the $z(t_1)$'s? By running them through an out-of-the-box ODE Solver. This brings up an interesting point that in order to even generate the dataset for these models to learn from, we need to utilize existing numerical solvers and incur the cost of those.

Our neural network f will model the dynamics themselves:

$$\frac{dz}{dt} = f(z(t), t, \theta) \tag{24.4}$$

So, with our dataset of trajectories, we can now formulate our optimization problem:

$$\begin{aligned}\min_\theta \mathcal{L}\left(z(t_1)\right) &= \mathcal{L}\left(z(t_0) + \int_{t_0}^{t_1} f(z(t), t, \theta) dt\right) \\ &= \mathcal{L}\left(\text{ODESolve}(z(t_0), f, t_0, t_1, \theta)\right)\end{aligned}$$

The forward pass is therefore straightforward: we pass our model of the dynamics and the initial condition $z(t_0)$ to an ODE solver, have it compute our desired output $z(t_1)$, and compare it to the true output at time t_1.

The backward pass raises an interesting question: in order to compute the gradient $\frac{d\mathcal{L}}{d\theta}$ to run the optimization, we need to run backpropagation through the ODE solver. How can we do this in a way that works for any choice of solver?

24.1.4 The Adjoint Sensitivity Method

In an ODE solver from $t_0 \to t_1$, we do not simply do do one step. Instead, we break up our trajectory into a series of much smaller steps; the smaller they are, the more accurate our solution, but at a higher computational cost.

Let's consider an intermediate step in this trajectory: $t_0 \to \cdots \to t \to \cdots \to t_1$, and call the evaluation at this step the **hidden state** $z(t)$. We can then analyze how the loss changes w.r.t this point, and refer to it as the **adjoint**:

$$a(t) = \frac{\partial \mathcal{L}}{\partial z(t)} \tag{24.5}$$

The dynamics of this adjoint happen to follow an ODE themselves:

$$\frac{da(t)}{dt} = -a(t)^\top \frac{\partial f(z(t), t, \theta)}{\partial z} \tag{24.6}$$

Why is this important? Because now we can use an ODE Solver to generate the $a(t)$'s! The initial condition is $a(t_1) = \frac{\partial \mathcal{L}}{\partial z_(t_1)}$, which we can solve for as it's simply the derivative of the loss function wrt the output of the

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

```
Input: dynamics parameters $\theta$, start time $t_0$, stop time $t_1$, final state $\mathbf{z}(t_1)$, loss gradient $\partial L/\partial \mathbf{z}(t_1)$
    $s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$          ▷ Define initial augmented state
    def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$):          ▷ Define dynamics on augmented state
        return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \theta}]$          ▷ Compute vector-Jacobian products
    $[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}]$ = ODESolve($s_0$, aug_dynamics, $t_1, t_0, \theta$)          ▷ Solve reverse-time ODE
return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$          ▷ Return gradients
```

FIGURE 24.3: Backpropagation via Adjoint Sensitivity Method with only one call an ODE solver

forward ODE solver. Then, we can run the dynamics with an ending goal in this backwards pass is to obtain $a(t_0) = \frac{\partial \mathcal{L}}{\partial z_(t_0)}$.

It's worth mentioning that the dynamics of $a(t)$ rely on knowing the hidden states $z(t)$ going backward in time. However, this is easy to do as we can just reverse the dynamics by starting at $z(t_1)$ going to $z(t_0)$.

With these, we can now solve for the derivative that we need to optimize our model:

$$\frac{d\mathcal{L}}{d\theta} = -\int_{t_1}^{t_0} a(t)^\top \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt \tag{24.7}$$

What's nice is that both Equation 24.6 and 24.7 above have a vector-Jacobian product, which can be efficiently done with any auto-differentiation software.

From an implementation perspective, we have 3 integrals to obtain the necessary values: $z, a, \frac{d\mathcal{L}}{d\theta}$. By concatenating these in a particular way, we can do these all in one call to an ODE solver, saving computation time. The details can be found in Figure 24.3

Thus, we have a way of backpropagating the error without having to worry about our choice of ODE solver. A nice diagram illustrating these forward and backward passes can be found in Figure 24.4 In summary, we have found a way to formulate a neural network, dubbed a **Neural ODE**[1], to model the dynamics of a system and train it on a variety of systems using standard machine learning optimization methods.

24.1.5 Latent Neural ODE's

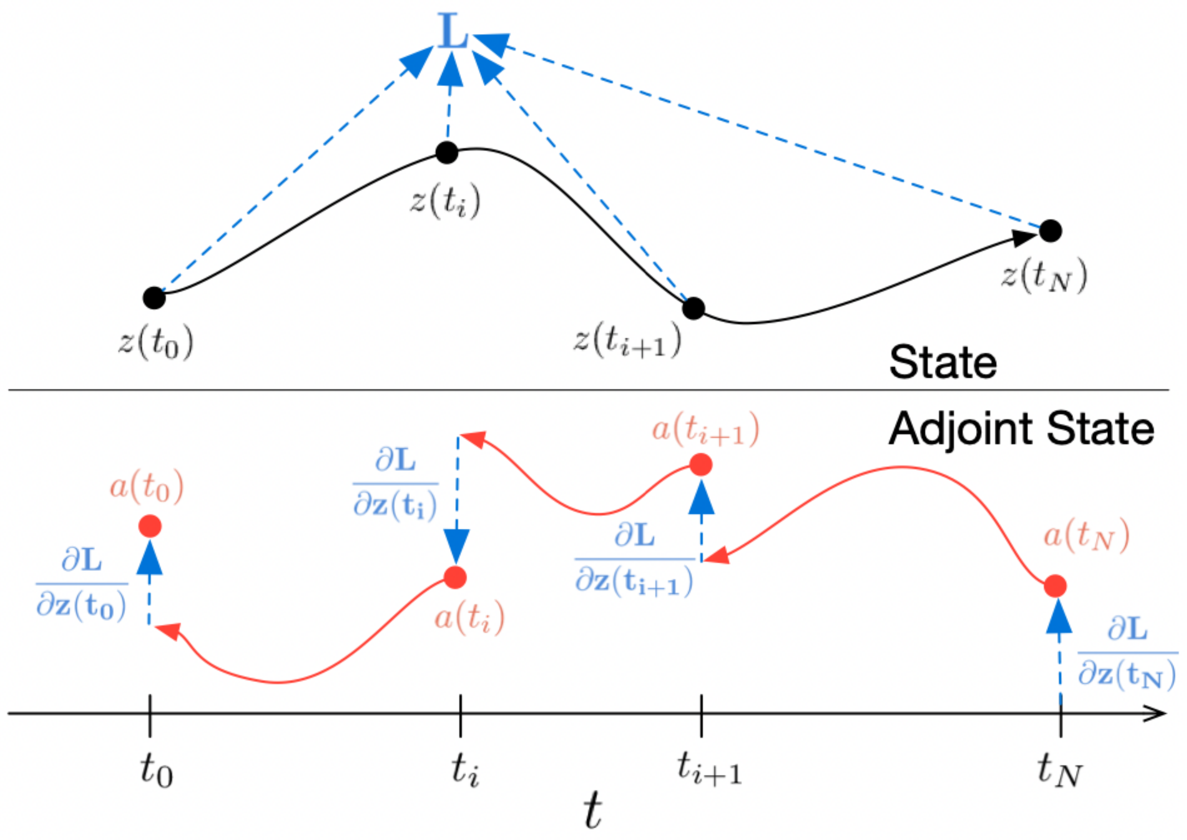
One creative way of using these Neural ODE's is to apply them to time series data, where it is much more common for there to be missing data at random intervals. The idea is to have the ODE model the dynamics of a series of **latent states** z_t.

The setup is similar to the VAE algorithms we have seen before: the time series inputs $x_1, \ldots, x_N$ are passed through a model suited for sequential data, e.g. a Recurrent Neural Network, and generate the parameters for a Gaussian prior that models a latent initial condition z_0. Then, the decoding process is analagous to running an ODE solver, with our underlying learnable dynamics f_θ.

$$\begin{aligned} z_0 &\sim p(z_0) \approxeq q_\phi(z_0|x_1, \ldots, x_N) \\ z_1, \ldots, z_N &= \text{ODESolve}(z_0, f, \theta_f, t_0, \ldots, t_N) \\ x_i &\sim p(x|z_i, \theta_x) \end{aligned}$$

A visualization of this process can be found in Figure 24.5

Interestingly, we are not limited to the N input datapoints. The continuous nature of an ODE solver allows us to sample the 'trajectory' of the time series data at a higher granularity, allowing us to extrapolate from



L
$z(t_i)$
$z(t_0)$
$z(t_{i+1})$
$z(t_N)$
State
Adjoint State
$a(t_0)$
$\frac{\partial L}{\partial z(t_0)}$
$\frac{\partial L}{\partial z(t_i)}$
$a(t_i)$
$a(t_{i+1})$
$\frac{\partial L}{\partial z(t_{i+1})}$
$a(t_N)$
$\frac{\partial L}{\partial z(t_N)}$
t_0
t_i
t_{i+1}
t_N
t

FIGURE 24.4: The forward and reverse dynamics that allow our Neural ODE to run backpropagation while being agnostic to the ODE solver used

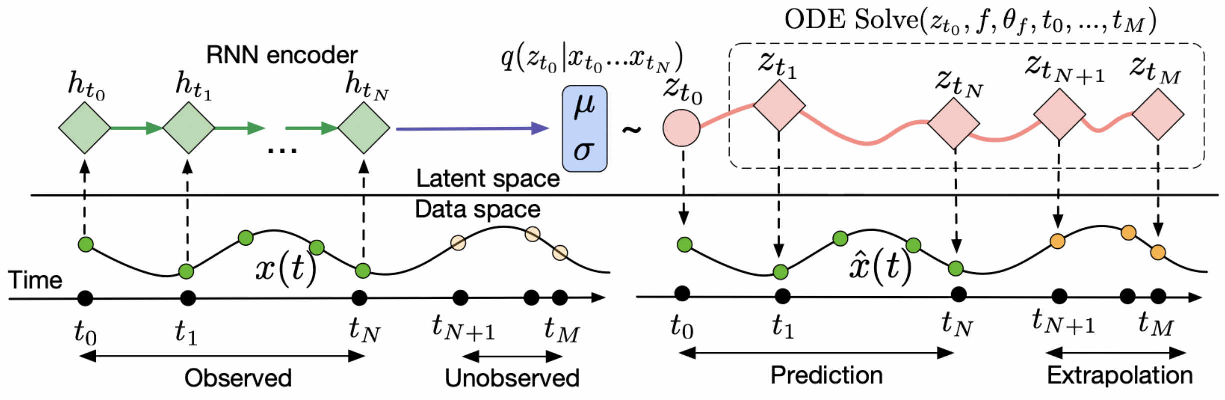


FIGURE 24.5: Latent ODE model for generative time series modelling

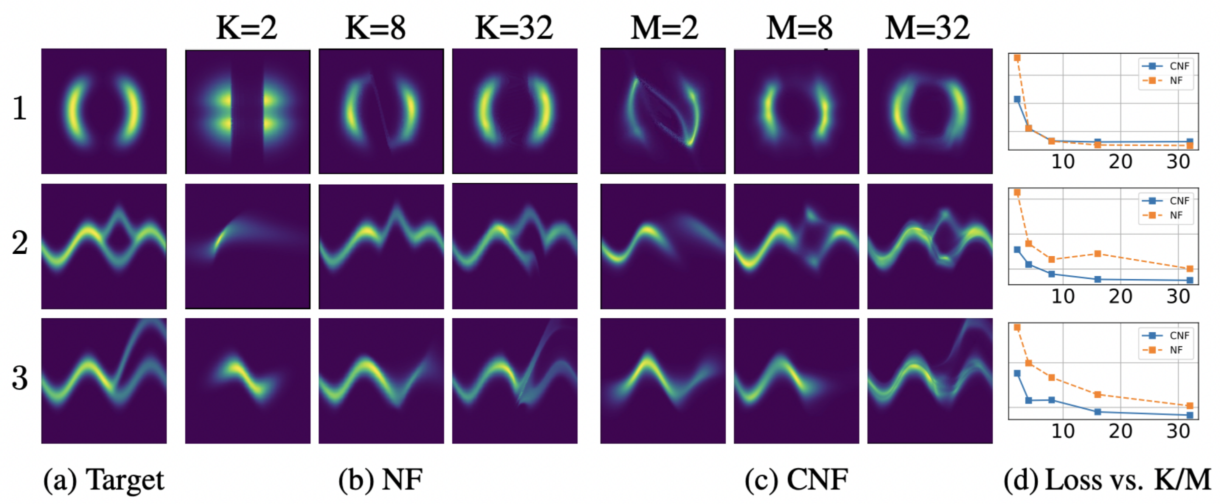


FIGURE 24.7: Comparing standard Normalizing Flows (NF) to Continuous Normalizing Flows (CNF). N is the depth of the NF, while M is the width of the CNF.

the given data. As this formulation fits into the VAE formulation, we can train this process by maximizing the ELBO. We can see the efficacy of this method to interpolate unseen points in Figure 24.6.

24.1.6 Continuous Normalizing Flows

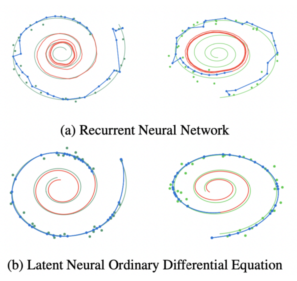


FIGURE 24.6: The results of a Recurrent Neural Network vs. the Latent Neural ODE in reconstructing and interpolating a spiral

Now that we have the ability to model solving an ODE with a neural network, we can actually revisit a topic that we have seen with a better attack strategy. Recall that in a normalizing flow model, such as NICE[2], an input is iteratively transformed via a carefully designed function whose Jacobian determinant must be tractable:

$$\log p(z_1) = \log p(z_0) - \log \left| \det \frac{\partial f}{\partial z_0} \right| \tag{24.8}$$

If we allow ourselves to consider the transformation to be continuous instead of a finite number of discrete layers, then the following theorem holds:

Theorem 1 (Instantaneous Change of Variables)**.** *Let $z(t)$ be a finite continuous random variable with probability $p(z(t))$ dependent on time. Let $\frac{dz}{dt} = f(z(t), t)$ be a differential equation describing a continuous-in-time transformation of $z(t)$. Assuming that f is uniformly Lipschitz continuous in z and continuous in t, then the change in log probability also follows a differential equation,*

$$\frac{\partial \log p(z(t))}{\partial t} = -Tr\left(\frac{df}{dz(t)}\right) \tag{24.9}$$

Thus, we can now model the dynamics of the probability density with just a trace instead of the determinant of a Jacobian. In practice, we can sample from $p(z_0)$, run the dynamics, and sample and get the density from $p(z_t)$. As we can see in Figure 24.7, these CNF's perform well compared to NF's.

24.1.7 FFJORD

The Free-form Jacobian of Reverseible Dynamics (FFJORD) algorithm is an approach introduced in [4] to model some normalizing flow that estimates some unknown distribution $p(\mathbf{x})$. Formally, the KL divergence between $p(\mathbf{x})$ and some parameterized distribution $p_\theta(\mathbf{x})$ is minimized,

$$\ell(p_\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(\mathbf{x}_i), \tag{24.10}$$

where N is the number of random samples drawn from the true distribution.

The approximate distribution $p_\theta(\mathbf{x})$ is parameterized using a learned vector field,

$$\mathbf{f}(\mathbf{x}, t) : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d. \tag{24.11}$$

Tracing out a path along the vector field gives an ODE,

$$\frac{d\mathbf{z}}{dt} = f(\mathbf{z}, t; \theta), \tag{24.12}$$

where we set the initial condition $\mathbf{z}(0) = \mathbf{x}$. Denoting $\mathbf{z}(\mathbf{x}, t; \theta)$ as the flow map evaluating the solution of the ODE at time t, we can write the log probability of our parametric distribution like

$$\log p_\theta(\mathbf{x}) = \log q(\mathbf{z}(\mathbf{x}, t; \theta)) + \log\det\left|\nabla\mathbf{z}(\mathbf{x}, t)\right|, \tag{24.13}$$

where q is a known distribution at time t, for example a Gaussian. Evaluating the determinant of the Jacobian can be troublesome, however, and so we employ a trick from fluid mechanics to rewrite this in the integral form,

$$\log p_\theta(\mathbf{x}) = \log q(\mathbf{z}(\mathbf{x}, t; \theta)) + \int_0^t \operatorname{div}(\mathbf{f})(\mathbf{z}(\mathbf{x}, \hat{t}; \theta), \hat{t}; \theta) d\hat{t}. \tag{24.14}$$

24.1.8 Regularization

A drawback of the regular FFJORD method is that the learned vector field that minimizes eq. (24.10) is often not unique; in particular if one observes fig. 24.8, there can be ill-conditioned vector fields that take meandering paths to move between distributions.

If the integral is being solved numerically, then this can introduce additional problems when computing the flow through the vector field. An ODE that contains rapidly varying derivative terms is often referred to as *stiff* [5], and these can lead to unstable numerical solvers unless the step size is taken to be very small. As a small step size implies a large time to solution, it is in our interest to make sure the vector field is well conditioned and does not contain large variations.

The paper motivates regularizing the vector field[3] by measuring the force experience by a particle in the flow under the vector field dynamics, which they do by taking the derivative with respect to time,

$$\frac{d\mathbf{f}(\mathbf{z}, t)}{dt} = \langle \nabla\mathbf{f}(\mathbf{z}, t), \mathbf{f}(\mathbf{z}, t)\rangle + \frac{\partial f(\mathbf{z}, t)}{\partial t}. \tag{24.15}$$

Well conditioned flows should have low acceleration, instead opting to have nearly constant force on particles as they travel through the vector field; penalty terms are thus placed on both $\mathbf{f}$ and $\nabla\mathbf{f}$.

24.1.8.1 Optimal Transport

This problem of moving between two densities can be modelled as an optimal transport problem between the densities $p(\mathbf{x})$ and $q(\mathbf{x})$ using a map $\mathbf{z} : \mathbb{R}^d \to \mathbb{R}^d$, minimizing

$$M(\mathbf{z}) = \int \|\mathbf{x} - \mathbf{z}(\mathbf{x})\|^2 p(\mathbf{x}) d\mathbf{x}, \tag{24.16}$$

subject to the constraint that

$$\int_A q(\mathbf{z}) d\mathbf{z} = \int_{\mathbf{z}^{-1}(A)} p(\mathbf{x}) d\mathbf{x}, \tag{24.17}$$

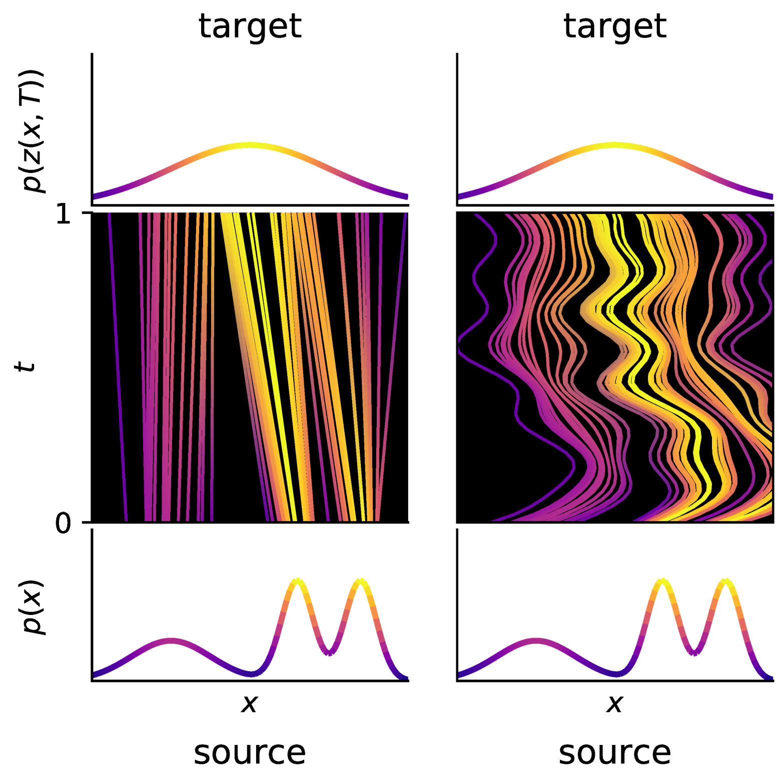


FIGURE 24.8: (Left) Optimal transport map, and (right) a generic normalizing flow. Both transport values from one distribution to another, but the right takes a suboptimal path.

meaning the measure of any arbitrary set A is preserved under $\mathbf{z}$.

This can alternatively be done by minimizing

$$\min_{\mathbf{f},\rho} \quad \int_0^t \int \|\mathbf{f}(\mathbf{x},t)\|^2 \rho_t(\mathbf{x}) d\mathbf{x} dt, \tag{24.18}$$

$$\text{st} \quad \frac{\partial \rho_t}{\partial t} = -\operatorname{div}(\rho_t \mathbf{f}), \tag{24.19}$$

$$\rho_0(\mathbf{x}) = p, \tag{24.20}$$

$$\rho_t(\mathbf{z}) = q, \tag{24.21}$$

where the above measures the *kinetic energy* of the flow. The first constraint ensures conservation of mass, while the last two force the learned distribution to agree with p and q. We can use this key intuition from optimal transport to regularize the vector field we get. Thus, we can rewrite the objective function to get

$$J_\lambda(\mathbf{f}) = \frac{\lambda}{N} \sum_{i=1}^N \int_0^t \|\mathbf{f}(\mathbf{z}_i, t\|^2 dt - \frac{1}{N} \sum_{i=1}^N \log p_\theta(\mathbf{x}_i), \tag{24.22}$$

where λ is a constant weight term.

24.1.9 Frobenius Regularization of Jacobian

In order to force the expression in eq. (24.15) to be small overall, we also need a regularization term on the Jacobian, $\nabla \mathbf{f}$. Recall that the Frobenius norm of some arbitrary matrix $\mathbf{A}$ can be written in the following equivalent ways:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i,j} a_{ij}^2} = \sqrt{\operatorname{tr}(\mathbf{A}\mathbf{A}^T)} = \sqrt{\operatorname{tr}(\mathbf{A}^T\mathbf{A})} \tag{24.23}$$

Furthermore, we can estimate the trace of some $\mathbf{B} \in \mathbb{R}^{n\times n}$ by taking a quadratic product with random vectors,

$$\operatorname{tr}(\mathbf{B}) = \mathbb{E}_{\mathbf{z}\sim\mathcal{N}(0,1)} \left[\mathbf{z}^T \mathbf{B} \mathbf{z}\right]. \tag{24.24}$$

Using this, we can estimate the Frobenius norm of $\mathbf{A}$ using the definition above to get

$$\|\mathbf{A}\|_F^2 = \mathbb{E}_{\mathbf{z}\sim\mathcal{N}(0,1)} \left[(\mathbf{A}\mathbf{z})^T(\mathbf{A}\mathbf{z})\right] = \mathbb{E}_{\mathbf{z}\sim\mathcal{N}(0,1)} \left[(\mathbf{z}^T\mathbf{A})(\mathbf{A}^T\mathbf{z})\right]. \tag{24.25}$$

Thus, we can estimate the Frobenius norm of the Jacobian like

$$\|\nabla \mathbf{f}\|_F^2 = \mathbb{E}_{\mathbf{z}\sim\mathcal{N}(0,1)} \left[(\mathbf{z}^T \nabla \mathbf{f}(\mathbf{z}))(\mathbf{z}^T \nabla \mathbf{f}(\mathbf{z}))^T\right]. \tag{24.26}$$

The $\mathbf{z}^T \nabla \mathbf{f}(\mathbf{z})$ term can be computed efficiently using reverse-mode automatic differentiation, thus approximating the Frobenius norm can be done for low computational cost.

24.1.10 Regularized Optimization

We can modify the original FFJORD objective function to include all of the above regularization penalties and obtain the final optimization objective,

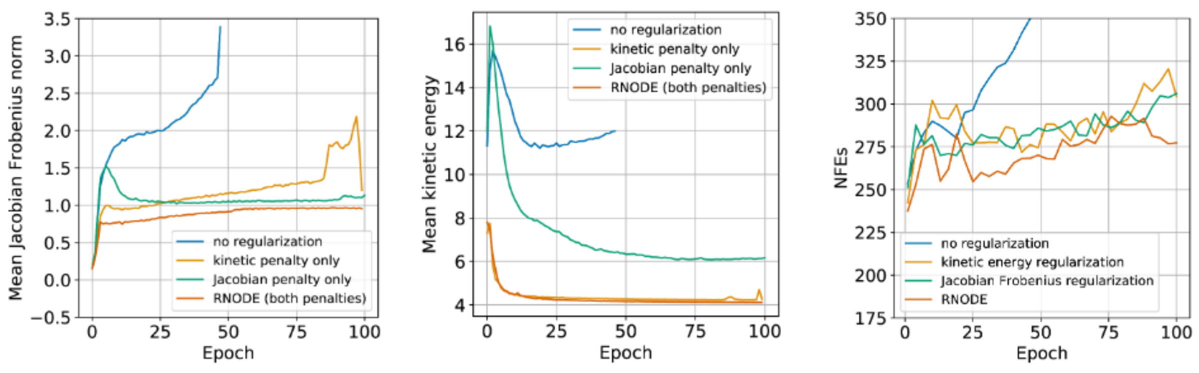
$$\begin{aligned} \ell = & \frac{1}{Nd} \sum_{i=1}^N - \log q(\mathbf{z}(\mathbf{x}_i, t)) - \int^t \operatorname{div}(\mathbf{f})(\mathbf{z}(\mathbf{x}_i, s; \theta) s; \theta) ds + \\ & \lambda_K \int_0^t \|\mathbf{f}(\mathbf{z}(x_i, s; \theta), s, \theta\|^2 ds + \lambda_j \int_0^t \|\nabla_{\mathbf{z}} \mathbf{f}(\mathbf{z}(\mathbf{x}_i, s; \theta), s; \theta)\|_F^2 ds. \end{aligned} \tag{24.27}$$

The $\mathbf{z}(\mathbf{x}, t)$ term is the solution of an ODE, which we can solve for numerically by some numerical ODE solver. We can augment the ODE to get a larger system of differential equations, defined like

$$\frac{d\mathbf{z}}{dt} = \mathbf{f}(\mathbf{z}, t) \tag{24.28}$$
$$\frac{dl}{dt} = \operatorname{div}(\mathbf{f})(\mathbf{z}, t) \tag{24.29}$$
$$\frac{dE}{dt} = \|\mathbf{f}(\mathbf{z}, t)\|^2 \tag{24.30}$$
$$\frac{dn}{dt} = \|\nabla \mathbf{f}(\mathbf{z}, t)\|_F^2, \tag{24.31}$$

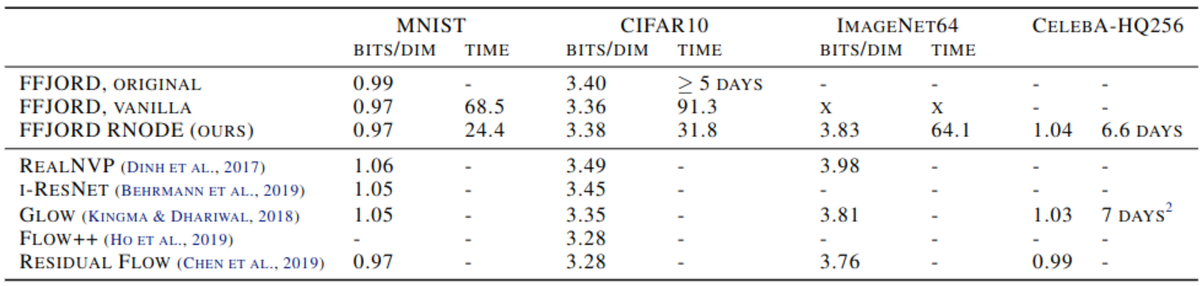
with initial conditions $\mathbf{z}(0) = \mathbf{x}$, $E(0) = l(0) = n(0) = 0$. The E term refers to the kinetic energy of the system, the l the log determinant of the Jacobian, and n the integral of the Frobenius norm of the Jacobian.

24.1.11 Results of Regularization



(a) Jacobian norm (b) Kinetic energy (c) Function evaluations

FIGURE 24.9: Ablation study of the effect of using the regularizers.



	MNIST		CIFAR10		IMAGENET64		CELEBA-HQ256	
	BITS/DIM	TIME	BITS/DIM	TIME	BITS/DIM	TIME		
FFJORD, ORIGINAL	0.99	-	3.40	$\geq$ 5 DAYS	-	-	-	-
FFJORD, VANILLA	0.97	68.5	3.36	91.3	X	X	-	-
FFJORD RNODE (OURS)	0.97	24.4	3.38	31.8	3.83	64.1	1.04	6.6 DAYS
REALNVP (DINH ET AL., 2017)	1.06	-	3.49	-	3.98	-	-	-
I-RESNET (BEHRMANN ET AL., 2019)	1.05	-	3.45	-	-	-	-	-
GLOW (KINGMA & DHARIWAL, 2018)	1.05	-	3.35	-	3.81	-	1.03	7 DAYS[2]
FLOW++ (HO ET AL., 2019)	-	-	3.28	-	-	-	-	-
RESIDUAL FLOW (CHEN ET AL., 2019)	0.97	-	3.28	-	3.76	-	0.99	-

FIGURE 24.10: Comparison of the standard FFJORD without regularization (ORIGINAL, VANILLA), and one with the above proposed regularization (RNODE). Also for comparison are several other flow implementations.

As can be seen in fig. 24.10, applying regularization to the vector field can drastically reduce the training time required for the FFJORD models. Additionally, an ablation study of using the regularization terms was performed on MNIST and detailed in fig. 24.9. Without any regularization, the dynamics are unstable and fail after about 50 training epochs, while adding any of the penalties greatly helps the kinetic energy and the dynamics of the vector field.

References

[1] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019.

[2] Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation, 2015.

[3] Chris Finlay, Jörn-Henrik Jacobsen, Levon Nurbekyan, and Adam M Oberman. How to train your neural ode: the world of jacobian and kinetic regularization, 2020.

[4] Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. Ffjord: Free-form continuous dynamics for scalable reversible generative models, 2018.

[5] M.T. Heath. *Scientific Computing: An Introductory Survey.* McGraw-Hill Education, 2005.