A Tutorial on Neural ODEs

Sharan G
sharan20051998@gmail.com

Consider a vanilla residual network [2], with each residual block proposing a difference that must be added to the input to change it, ultimately leading to the output. This could be viewed as a field, conditioned on the input itself, in discrete space (each countable layer), so in a way, the input is "placed" at the input "coordinate", and "moved through" the discrete space. At each point, the field proposes a difference that is added to the input. The training algorithm that we use helps the field "learn" by tweaking the parameters, so that the differences proposed at each layer leads to an output closer to the output we want (or to reduce the loss, for that matter).

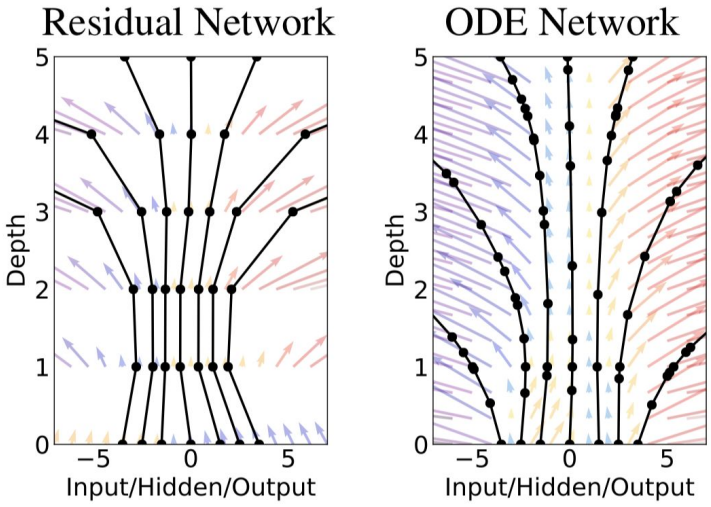
This particular field, in our case, is parameterized by an index number i, the layer number, and a set of parameters. It just so happens in this case, that only a subset of those parameters are used at each coordinate i (which is of course, are the parameters of that particular layer: θ_i which is a subset of θ which is the vector of parameters of the network), but we will consider a more general case. We write the following equation to describe the field:

$$\Delta x(i) = f(x(i), i; \theta)$$

If the network has layers indexed from 0 to $l-1$, the output of the layer i, denoted by $x(i+1)$ is obtained as

$$x(i+1) = x(i) + f(x(i), i; \theta)$$

where $x(0)$ is the input to the network and $x(l)$ is the output. Note that x refers to a d dimensional vector.



Source: https://arxiv.org/abs/1806.07366 [1]

The idea behind Neural ODEs [1] is that we use a continuous field, instead of a discrete one as above. So instead of a set of parameters used to model differences at discrete points (or layers), we model the derivatives at each point i in continuous space. We then utilize this derivative to find out the integral (analogous to the sum in the discrete case) of the function

$$\frac{dx}{dt} = f(x(t), t; \theta)$$

where x is a function of t. $x(0)$ is the input to the network, and the integral has to be evaluated over the continuous interval (t_0, t_1) where $t_1 - t_0$ is analogous to how deep the network was in the discrete case (number of residual units). The output is then

$$x(t_1) = \int_{t_0}^{t_1} f(x(t), t; \theta)\, dt$$

The above ideas and equations also apply to RNN units, where the stateful RNN layer can be designed to propose a change to the state rather than produce a new state itself. Instead of discrete inputs, we can simply integrate from the initial t_0 all over to t_1, to produce the output at time t_1.

A popular family of algorithms to evaluate integrals of the above form, that is:

$$x(t_1) = \int_{t_0}^{t_1} f(x(t), t)\, dt$$

which is the "solution" to

$$\frac{dx(t)}{dt} = f(x(t), t)\,,\ x(t_0) = x_0$$

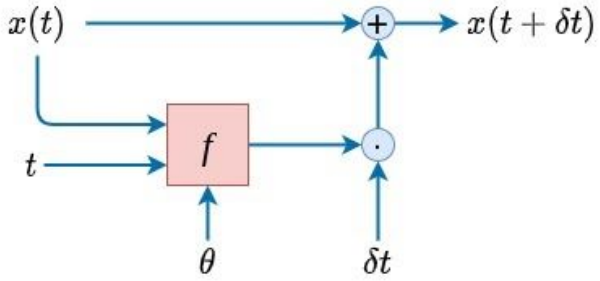
are that of Runge-Kutta Methods, the simplest of which is the Euler method to numerically evaluate such integrals, which uses a first order approximation of the function at each point, which means that the algorithm simply finds the derivative at each point $x(t)$ and takes a small step ε towards t_2, adding $\varepsilon \frac{dx(t)}{dt}$ to $x(t)$. This is the first order Runge Kutta method. The most popular one is the fourth order one. These numerical integral evaluation methods to approximate such integrals, or equivalently, to solve the ODEs are called ODE solvers. We can use any ODE solver, as the requirement is.

It might be unclear at his point as to finding out the derivatives of the parameters θ is this case, when an integral is being evaluated. 2 approaches are:

1) Simply record the operations that are being done by the ODE solver itself, and backpropagate gradients through those operations. This will work well enough to approximate derivatives.

2) Look at the integral operation above in small, infinitesimal chunks, pass back gradients through them and "add" contributions of each chunk, and then integrate. This is explained in detail in the next section, and is known as the adjoint sensitivity method. A convenient side product of this method is that we can carry out backpropagation with constant memory, unlike in the case of a regular neural network, where we need to save the forward operations ($O(l)$ in both time and space, where l is the number of layers), or at least checkpoints at some intervals in order to backpropagate.

The Adjoint Sensitivity Method



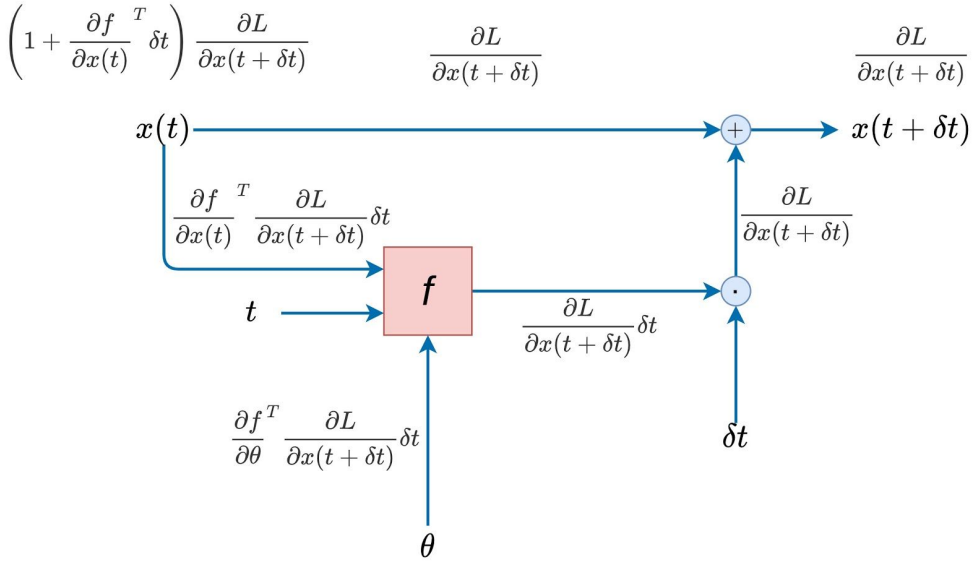
For an infinitesimal step δt, the following can be written:

$$x(t+\delta t) = x(t) + f(x(t), t; \theta) * \delta t$$

The computation graph for this small step is given on the left.

During backpropagation, assume that you have somehow backpropagated gradients till the point $t+\delta t$ which are gradients for $x(t+\delta t)$ and need to compute the gradients for $x(t)$ and θ (for this timestep).

Those can be computed using the chain rule as follows:



All derivatives backpropagated in this infinitesimal step operation have been shown above.

First of all, note that the final gradient for θ can be computed by summing the contributions from each of these operations, as the gradients ending at θ from one single infinitesimal step do not flow over any further, which means we have to simply sum up.

We will denote $\frac{\partial L}{\partial x(t+\delta t)}$ as a new function $a(t)$. This is called the adjoint function.

Hence, we have:

$$\Delta\theta = \lim_{\delta t \to 0} \sum_{t \text{ in } \{t_0 + n\delta t \mid 0 < n < \frac{(t_1 - t_0)}{\delta t}\}} \frac{\partial f}{\partial \theta}^T \frac{\partial L}{\partial x(t+\delta t)} \delta t$$

which gives:

$$\Delta\theta = \int_{t_0}^{t_1} \frac{\partial f}{\partial \theta}^T a(t)\delta t = -\int_{t_1}^{t_0} \frac{\partial f}{\partial \theta}^T a(t)\delta t$$

Note that we have not yet computed these $a(t)$ at each of these points. Moreover, the source from which the Neural ODE received the input $x(t_0)$ might expect the gradients with respect to the input.

Hence, we also have to compute $a(t)$.

At t,

$$a(t) = \frac{\partial L}{\partial x(t)} = \frac{\partial f}{\partial x(t)}^T \frac{\partial L}{\partial x(t+\delta t)} \delta t + a(t + \delta t) \qquad \text{(derivative to the input)}$$

$$\Rightarrow \frac{a(t+\delta t) - a(t)}{\delta t} = \frac{\partial f}{\partial x(t)}^T \frac{\partial L}{\partial x(t+\delta t)} \delta t$$

$$\Rightarrow \lim_{\delta t \to 0} \frac{a(t+\delta t) - a(t)}{\delta t} = \lim_{\delta t \to 0} \frac{\partial f}{\partial x(t)}^T \frac{\partial L}{\partial x(t+\delta t)} \delta t$$

$$\Rightarrow \frac{da(t)}{dt} = \frac{\partial f}{\partial x(t)}^T a(t)$$

Hence, the gradient of the loss with respect to the value that is being evaluated by the integral along the path of the integral (which is from t_0 to t_1) can be computed by:

$$\frac{\partial L}{\partial x(t_0)} = a(t_0) = \int_{t_0}^{t_1} \frac{\partial f}{\partial x(t)}^T a(t)\delta t = \quad -\int_{t_1}^{t_0} \frac{\partial f}{\partial x(t)}^T a(t)\delta t$$

where $a(t_1)$ will be given, as passed from the system after the NODE, which may be a simple loss function, or an entire differentiable function.

A convenient way to carry out these computations is by concatenating them together and passing to the black box ODE solver as one single vector. This will also ensure that each computation for θ has the corresponding $a(t)$ computed, which is required as given in the above equation that gives the integral of θ.

Another important point to note is that while backpropagating, we can simply recompute $x(t)$ in reverse! This will introduce some numerical error, but it practically works really well. Now we can setup the algorithm to work without needing to store forward computed values, unlike in the vanilla NN case.

Hence, the algorithm for a single input x at t_0, and given derivative with respect to the output $\frac{\partial L}{\partial x(t_1)}$ (or $a(t_1)$) is given by:

```
theta = parameter vector of the neural net
f(x, t; theta) = neural net function, parameterized by theta.
ode_solve(x_0, t_0, t_1, f) = black_box ode solver

def forward(x_0):     # returns x_1
   return ode_solve(x_0, t_0, t_1, f(_, _; theta))

def backward(x_1, a_1):     # returns del_a_0 and del_theta which are input derivative and parameter's derivatives

   concat_vector = [x_1, a_1, 0] #0 for computing theta's grads

   def concat_vector_derivative(vector, t):
         current_x, current_a, current_theta = vector
         del_f_theta, del_f_x = Jacobian of f(current_x, current_a, theta) w.r.t. [current_theta, current_x]
         return [f(current_x, current_t; theta), -transpose(del_f_x).current_a, -transpose(del_f_theta).current_a]

   x_0, a_0, delta_theta = ode_solve(concat_vector, t_1, t_0, concat_vector_derivative)
   return a_0, delta_theta
```

There are a multitude of examples provided in the paper, such as a time series model where the neural ODE is used to implement a VAE which models the dataset. We can trade-off speed for precision (more computations to compute integral => slower and vice versa).

References

1. [1806.07366] Neural Ordinary Differential Equations
2. [1512.03385] Deep Residual Learning for Image Recognition