Neural Ordinary Differential Equations for Memristor Crossbars

Louis Primeau

University of Toronto

Abstract

This document is an attempt at laying out all the concepts and work necessary for writing a paper on the implementation of neural ordinary differential equations for the crossbar. It will go over both theory and implementation of the adjoint method, ODE solvers for the crossbar, Fourier transforms, and Hadamard products.

Keywords: None.

Contents

1. Introduction

Neural Ordinary Differential Equations are a novel technique for representing the structure of neural networks, reducing the number of parameters needed to learn a problem. In a conventional neural network, we might have several convolutional layers in series through which an image flows and is successively transformed into a linearly separable phase space. On a crossbar, this means significant real estate is used to represent the many convolutions needed to do inference on complicated image datasets. To combat this, we can reduce the number of parameters by using neural ordinary differential equations, which use less convolutional layers applied more times, allowing us to emulate the depth of deeper convolutional networks while retaining the lightweight memory usage of shallower networks.

This document aims to illuminate the algorithm of neural differential equations from the basics in order to show exactly how it might be implemented over a real crossbar.

2. Ordinary Differential Equations

In this section we will go over the notation needed to construct the appropriate algorithms for neural differential equations. Because the crossbar is a new piece of computing technology, there are no compilers that can automatically differentiate our functions. Thus it is important to know exactly how to manipulate the commonly encountered functions in neural networks. First we will establish the notation that will be used throughout the document. Then we will write some results about differential equation solvers and optimization methods.

2.1. Calculus Review & Notation

Here are some results from multivariable calculus presented in a notation that I will use for the rest of this document.

Let $\mathbb{R}^n$ be the space of column vectors with n elements, denoted by a boldface, such as the vector $\boldsymbol{x}$. The dual space is $\mathbb{R}^{1\times n}$, which are row vectors with n elements. Similarly a matrix with n rows and m columns is in the space $\mathbb{R}^{n\times m}$.

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a function such that for $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{y} \in \mathbb{R}^m$,

$$\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{\theta}). \tag{1}$$

letting $\boldsymbol{\theta} \in \mathbb{R}^q$ be a vector of parameters. Then the partial derivative of f with respect to x_i, the ith entry of the vector $\mathbf{x}$ is denoted by

$$\frac{\partial f}{\partial x_i} \text{ or } \frac{\partial y}{\partial x_i} \in \mathbb{R}^{1\times m} \tag{2}$$

where the derivative is a row vector of length m. The Jacobian of f with respect to $\boldsymbol{x}$ is

$$\nabla_{\boldsymbol{x}} f = \left[\frac{\partial f}{\partial x_i}\right]_i \in \mathbb{R}^{n\times m} \tag{3}$$

where the Jacobian is just the partial derivative vectors stacked as rows in a big matrix in $\mathbb{R}^{n\times m}$. Note that this is distinct from the gradient with respect to the parameters $\boldsymbol{\theta}$:

$$\nabla_{\theta} f = \left[\frac{\partial f}{\partial \theta_i}\right]_i \in \mathbb{R}^{n\times q} \tag{4}$$

which has a number of columns equal to the number of parameters in $\boldsymbol{\theta}$. Let $f : \mathbb{R}^n \to \mathbb{R}^m$ and $g : \mathbb{R}^p \to \mathbb{R}^n$. We can compose them:

$$\boldsymbol{y} = g(\boldsymbol{z}, \boldsymbol{\phi}) = g \circ f(\boldsymbol{x}) = g(f(\boldsymbol{x}, \boldsymbol{\theta}), \boldsymbol{\phi}) \tag{5}$$

and then derive gradients with respect to the the various inputs. The chain rule can be demonstrated through the following calculation:

$$\nabla_{\boldsymbol{x}} \boldsymbol{y} = \nabla_{\boldsymbol{x}}(g \circ f(\boldsymbol{x})) = \underbrace{\nabla_{\boldsymbol{z}} g|_{f(\boldsymbol{x})}}_{p\times m} \overbrace{\nabla_{\boldsymbol{x}} f|_{\boldsymbol{x}}}^{m\times n} \underbrace{\cancelto{I_n}{\nabla_{\boldsymbol{x}} \boldsymbol{x}}}_{n\times n} \tag{6}$$

showing that it works just like the regular single variable derivative, except that you have to be careful of the dimension.

2.2. ODEs and ODE Solvers

In this application we are concerned with first order initial value problems (IVPs) where we wish to find z given its derivative

$$\frac{d\boldsymbol{z}}{dt} = f(\boldsymbol{z}(t), t), \tag{7}$$

where f is a function that specifies the dynamics of z through time, i.e. its derivative. $\boldsymbol{z}$ is an implicit function of time, so its dependence on t will be assumed throughout, and $z(t)$ will be written as z^t. The function is specified at a time t_0, thus we have $f(\boldsymbol{z}(t_0), t_0) = \boldsymbol{z}^0$ This problem may be solved through the integration

$$\boldsymbol{z} = \int_{t_0}^{t_f} f(\boldsymbol{z}, t)dt + \boldsymbol{z}^0 \tag{8}$$

which is what we have to approximate with our ODE solver. The simplest ODE solver is the first-order Euler's Method, which makes the following approximation to the integral at every timestep:

$$\boldsymbol{z}^{n+1} = f(\boldsymbol{z}^n, t) \times h + \boldsymbol{z}^n. \tag{9}$$

Letting the superscript denote the timestep of $\boldsymbol{z}$. A more stable ODE solver is implicit Euler, which has the form

$$\boldsymbol{z}^{n+1} = f(\boldsymbol{z}^{n+1}, h) \times h + \boldsymbol{z}^n. \tag{10}$$

$\boldsymbol{z}^{n+1}$ is contained implicitly in the timestepping scheme. This means that we have to solve this algebraic equation. Since our function f will be nonlinear, we will use Newton's method, which finds the roots of a function $F(\boldsymbol{z})$ through the iterative scheme

$$\boldsymbol{z}_{n+1} = \boldsymbol{z}_n + (\nabla_{\boldsymbol{z}} F|_{\boldsymbol{z}_n})^{-1} F(\boldsymbol{z}_n). \tag{11}$$

Here it is very important to note that these $\boldsymbol{z}$s have subscripts, not superscripts. Here, and in the rest of the document, a subscript on a bold variable will mean iteration of Newton's Method. Applying this formula to equation (10), we get

$$F \equiv -\boldsymbol{z}^{n+1} + f(\boldsymbol{z}^{n+1}, h) * h + \boldsymbol{z}^n = 0 \tag{12}$$

$$\boldsymbol{z}_{n+1} = \boldsymbol{z}_n + (\nabla_{\boldsymbol{z}^{n+1}} F|_{\boldsymbol{z}_n})^{-1} F(\boldsymbol{z}^{n+1} = \boldsymbol{z}_n). \tag{13}$$

The trouble with implicit Euler is that Newton's method or equivalent minimization algorithms are extremely computationally expensive. It is worth experimenting with explicit methods first to see if the problem can be solved without having to resort to implicit methods.

3. Convolutional Neural Networks

Convolutional Neural Networks work on the principle of filtering, in which an input is successively convolved with learnable filters which transform the image into one that is tractable for classification or regression. A typical convolutional network is comprised of several types of layers:

1. A convolutional layer, which is a linear operation.
2. A maxpooling layer, which takes blocks of inputs and outputs their sum. As an example, it might take an 8×8 image and return a 4×4 image, where the image was sliced into 16 blocks of equal size, and the max of each of these was taken as an output.

3. An averagepooling layer, which does the same thing as the maxpooling layer but computes the average instead. Note that this type of layer is a special case of convolutions, except the kernel is not learnable.
4. A dropout layer, which randomly zeros some inputs. This layer is only used during training, to prevent the network from overfitting.
5. An element-wise nonlinear function, such as relu.

3.1. Convolutions

A convolution takes an image matrix and convolves its input with a kernel, which is another matrix. The typical explanation is a "sliding window", where the kernel is applied to a $m \times m$ area, then moved to the right a specified distance, and then applied again. "Applied" means that we take the Hadamard (element-wise) product of the kernel matrix and the part of the image it is acting over. A better explanation of convolutions can be found in one of the many sources online. Some properties are noted here, for completeness.

Convolutions are linear operations. To see this, we can consider an image vector with n pixels. For such an image a convolution can be represented as an $n \times n$ matrix. Below is the code to construct such a matrix in Python, for a 3×3 kernel with stride 1.

```python
def convolve(image, kernel):
    permuter = np.arange(image.size).reshape((int(image.size**0.5),)*2)
    directions = [np.roll(permuter, (-1,-1), axis=(0,1)), #sw
                  np.roll(permuter, (0,-1) , axis=(0,1)), #w
                  np.roll(permuter, (1,-1) , axis=(0,1)), #nw
                  np.roll(permuter, (-1,0) , axis=(0,1)), #s
                  permuter, #c
                  np.roll(permuter, (1,0)  , axis=(0,1)), #n
                  np.roll(permuter, (-1,1) , axis=(0,1)), #se
                  np.roll(permuter, (0,1)  , axis=(0,1)), #e
                  np.roll(permuter, (1,1)  , axis=(0,1))] #ne
    return sum([np.identity(image.size)[directions[i].reshape(image.size,1)] *
   kernel[i] for i in range(0,9)]).reshape(image.size, image.size)

```

Here we will make a distinction between a convolution kernel a, which in this code is a 3×3 array, and its convolution matrix A, which is the expanded form created by this code. The convolution matrix A serves to demonstrate the linearity of convolution operations. This is especially nice since it makes the derivation of the derivative very easy. For a convolution matrix A and image vector $\boldsymbol{x}$, the gradient w.r.t $\boldsymbol{x}$ is just

$$\nabla_{\boldsymbol{x}} A\boldsymbol{x} = A \nabla_{\boldsymbol{x}} \boldsymbol{x} = A. \tag{14}$$

This way to represent convolutions is mathematically useful but foolish to implement in practice. Another way to represent convolutions is in the Fourier domain, where a convolution is a Hadamard Product, i.e. element wise multiplication. The 2D fourier transform of an $n \times m$ matrix A is given by

$$\hat{A} = W_n A W_m \tag{15}$$

$$\mathbb{R}^{m\times m} \ni W_m = \left[\frac{\omega^{jk}}{\sqrt{m}}\right]_{j,k}, \ \omega = e^{-2\pi i/N} \tag{16}$$

Essentially, to get a 2D fourier transform matrix, you just transform the rows and the columns. This is nice because the convolution of the kernel a with the image matrix X (note the difference in representation) becomes

$$a * X = \hat{a} \otimes \hat{X} \tag{17}$$

where $\otimes$ denotes the Hadamard product of $\hat{a}$ and $\hat{X}$ and $*$ denotes the convolution. Since the crossbar can only represent real numbers on a single memristor, we have to use the following representation of complex numbers:

$$a + bi \to \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \tag{18}$$

which you can verify for yourself. Equation (16) then becomes

$$\mathbb{R}^{m\times m} \ni W_m = \left[\frac{\omega^{jk}}{\sqrt{m}}\right]_{j,k}, \; \omega = \begin{bmatrix} \cos(-2\pi/N) & \sin(2\pi/N) \\ \sin(-2\pi/N) & \cos(2\pi/N) \end{bmatrix} \tag{19}$$

3.2. ResNet

A Residual Neural Network (ResNet) is a specific type of convolutional network architecture, which is comprised of many residual blocks. Each residual block has six convolutional layers, and what are called skip connections, in which the output of a layer is added to the image's state before it entered that layer. See (He. et Al 2015) for details.

4. Gradient Descent

Gradient descent is one of the most basic optimization methods, in which in order to minimize a scalar loss function, we calculate the gradient w.r.t. to the loss and proceed a step in the opposite direction. Gradient descent is used in machine learning because the problems are highly non-convex and nonlinear[3], where other optimization techniques prove ineffective.

We can make a distinction between stochastic gradient descent and batch gradient descent. In stochastic gradient descent, we select randomly from our labels, compute the gradient, and update our parameters. Thus the optimization method is a stochastic process which tends towards the solution. In batch gradient descent, we compute the gradient for all labels, average them and then proceed in that direction. For each iteration, we might expect that batch gradient descent provides a better step than stochastic gradient descent at the cost of computation time. Batch methods, however, benefit from increased parallelizability.

Gradient Descent for machine learning is not especially sensitive to numerical accuracy. [4] shows that machine learning using 16-bit precision performs no worse than the same network architecture trained with 32-bit fixed-point representation. Even a 12-bit network performs performs only slightly worse than the full precision network. This however, depends on the rounding scheme that is used.

5. Neural Ordinary Differential Equations

Neural Ordinary Differential Equations are an alternative representation of neural networks. The architecture of ResNet, with the skip connections, suggests an euler-forward discretized neural network. With our knowledge of ODE solvers, we claim we can do better than euler forward, opting instead to use some sort of highly optimized adaptive method. On the crossbar, however, we don't have already built functions for solving arbitrary ODEs. This section discusses how to build an ODE solver for a function that we might use for Neural Ordinary equations. Then, we will derive the adjoint method, which we can use to calculate gradients w.r.t. the ODE solver we derive here.

5.1. ODE Solvers

The forward function of a neural ordinary differential equation is necessarily nonlinear, so that the network can learn nonlinear functions. Not only that, but the problem is stiff, such that we need an implicit solver. In the original paper, they use a 2nd order implicit Adams method. Here, I will use the 1st order implicit Adams method, also known as implicit Euler, due to limited space on the crossbar and ease of implementation. To combine all the previous parts of the paper, I will show the equations for the following setup.

```
f = conv(relu(conv(x, a)), b)
```

or

$$f(\boldsymbol{z}) = Br(A\boldsymbol{z}) \tag{20}$$

where $r(x)$ is the relu function. f has the derivative

$$\nabla_{\boldsymbol{z}} f = B\text{diag}(u(A\boldsymbol{z}))A \tag{21}$$

where $u(x)$ denotes the Heaviside step function, and diag places the elements of $A\boldsymbol{z}$ along the diagonal of a matrix. Then the differential equation for the neural network has the form

$$\frac{d\boldsymbol{z}}{dt} = B\text{relu}(A\boldsymbol{z}) \tag{22}$$

If we use implicit Euler to solve this equation, we have the timestepping scheme

$$\boldsymbol{z}^{n+1} = \boldsymbol{z}^n + hB\text{relu}(A\boldsymbol{z}^{n+1}) \tag{23}$$

which must be solved by Newton's method, which, as per above, leads to the following sub-iteration scheme

$$F(\boldsymbol{z}^{n+1}) \equiv -\boldsymbol{z}^{n+1} + \boldsymbol{z}^n + hB\text{relu}(A\boldsymbol{z}^{n+1}) \tag{24}$$

$$\nabla_{\boldsymbol{z}^{n+1}} F = -I + hB\text{diag}(u(A\boldsymbol{z}^{n+1}))A \tag{25}$$

$$\boldsymbol{z}_{n+1} = \boldsymbol{z}_n + (\nabla_{\boldsymbol{z}^{n+1}} F|_{\boldsymbol{z}_n})^{-1}[F|_{\boldsymbol{z}_n}] \tag{26}$$

$$\boldsymbol{z}_{n+1} = \boldsymbol{z}_n + (hBu(A\boldsymbol{z}_n)A - I)^{-1}[-\boldsymbol{z}_n + \boldsymbol{z}^n + hB\text{relu}(A\boldsymbol{z}_n)] \tag{27}$$

in which each $\boldsymbol{z}_i$ is a (hopefully) successively better approximation of z^{n+1}. This expression is what has to be mapped onto the crossbar. If we want to do training on the crossbar, we will have to apply the procedure here to Equation (36).

5.2. Adjoint Sensitivity Method

The algorithm given in the paper Neural Ordinary Differential Equations (2018) is mysterious and poorly explained. Other sources go into depth but are too complicated for our purposes. Here I will derive the adjoint sensitivity method for the simple case of an euler forward solver. This will make it clear where the algorithm comes from and its natural extension to general ODE solvers.

The premise of Neural ODEs is that the dynamics of the image vector $\boldsymbol{z}$ as it is transformed into a filtered version is governed by a first order ODE, given by

$$\frac{d\boldsymbol{z}}{dt} = f(\boldsymbol{z}(t), t, \boldsymbol{\theta}) \tag{28}$$

subject to

$$\boldsymbol{z}(0) = \boldsymbol{z}^0 \tag{29}$$

f where $\boldsymbol{z} \in \mathbb{R}^m$ and $f : \mathbb{R}^m \to \mathbb{R}^m$, supposing that the image $\boldsymbol{z}$ has m pixels. The euler discretization of this scheme is given by the discreet differential equation

$$\boldsymbol{z}^{n+1} = (I + \Delta t J^n)\boldsymbol{z}^n \tag{30}$$

and therefore the final state $\boldsymbol{z}^n$ is found by

$$\boldsymbol{z}^n = \prod_{i=0}^{n}(I + \Delta t J^i)\boldsymbol{z}^0 \tag{31}$$

where $J^i = \nabla_{\boldsymbol{z}} f|_{\boldsymbol{z}^i}$ is the Jacobian of f wrt to $\boldsymbol{z}$ evaluated at timestep i and is in effect the linearization of the differential equation in (28). We might represent the Jacobian of f by a series of convolutions, or as in [1], a standard residual block. Since we want to do inference, we construct additional functions $\Theta : \mathbb{R}^m \to \mathbb{R}$ which take our transformed image and output a loss. So our full neural network is

$$\mathcal{L} = \Theta\left(\prod_{i=0}^{n}(I + \Delta t J^i)\boldsymbol{z}^0\right). \tag{32}$$

Now we want to modify the parameters $\boldsymbol{\theta}$, so we need the gradient. Taking the gradient w.r.t $\boldsymbol{\theta}$ of (32) on both sides, we have

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\mathcal{L} &= \nabla_{\boldsymbol{\theta}}(\Theta(\prod_{i=0}^{n}(I + \Delta t J^i)\boldsymbol{z}^0)) \\ &= \nabla\Theta|_{\boldsymbol{z}^n}\nabla_{\boldsymbol{\theta}}(\prod_{i=0}^{n}(I + \Delta t J^i)\boldsymbol{z}^0)) \\ &= \nabla\Theta|_{\boldsymbol{z}^n}\sum_{i=0}^{n}([I + \Delta t J^n][I + \Delta t J^{n-1}]\dots\nabla_{\boldsymbol{\theta}}J^i\dots[I + \Delta t J^1])\boldsymbol{z}^0\end{aligned} \tag{33}$$

We could just compute the gradients through the equation derived in (33), but this would be highly computationally inefficient. Instead we can solve adjoint equation by taking the transpose of both sides

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}^T = (\boldsymbol{z}^0)^T\sum_{i=0}^{n}([I + \Delta t J^1]^T\dots(\nabla_{\boldsymbol{\theta}}J^i)^T\dots[I + \Delta t J^n]^T)\nabla\Theta^T|_{\boldsymbol{z}^n} \tag{34}$$

The quantity on the left side of (34) is called the adjoint. What's so special about the adjoint? Note that this looks very much like the Euler Forward scheme with three differences:

1. We seem to be running n different euler forward schemes (on account of the sum).
2. The index of the matrices is being run "backwards", from 1 to n instetad of n to 1
3. Partway through our euler forward run we apply the Jacobian of J^i instead of $I + \Delta t J^i$.

Out of these, 1. seems to be a big problem, since running n euler forwards would be horribly inefficient. In fact, this sum turns out to be only one euler forward of a new differential equation that we will construct. Consider the discreet differential equation

$$\begin{aligned}\boldsymbol{z}^{n+1} &= \boldsymbol{z}^n + \Delta t J^n\boldsymbol{z}^n + \nabla_{\boldsymbol{\theta}}J^n \\ &= [I + \Delta t J^n]\boldsymbol{z}^n + \nabla_{\boldsymbol{\theta}}J^n\end{aligned} \tag{35}$$

subject to the initial condition $y_0 = 0$. Let's construct the first couple terms:

$$\begin{aligned}
\boldsymbol{z}^1 &= \nabla_{\boldsymbol{\theta}} J^1 \\
\boldsymbol{z}^2 &= [I + \Delta t J^1]\nabla_{\boldsymbol{\theta}} J^1 + \nabla_{\boldsymbol{\theta}} J^2 \\
\boldsymbol{z}^3 &= [I + \Delta t J^2][I + \Delta t J^1]\nabla_{\boldsymbol{\theta}} J^1 + [I + \Delta t J^2]\nabla_{\boldsymbol{\theta}} J^2 + \nabla_{\boldsymbol{\theta}} J^3
\end{aligned}$$

That looks like equation (34), but we are missing the terms to the right of the gradient. Thus the appropriate construction of the adjoint problem is the block matrix equation

$$\begin{bmatrix} \boldsymbol{z}^{n+1} \\ \boldsymbol{q}^{n+1} \end{bmatrix} = \begin{bmatrix} I + \Delta t J^n & \nabla_{\boldsymbol{\theta}} J^n \\ \boldsymbol{0} & I + \Delta t J^n \end{bmatrix} \begin{bmatrix} \boldsymbol{z}^{n} \\ \boldsymbol{q}^{n} \end{bmatrix} \tag{36}$$

where we've added this extra term $\boldsymbol{q}$ to keep track of the terms before the gradients that were missing before. In the case of euler forward, this involves saving the state of the system after each time step, but fancier solvers will provide a function to interpolate the state of the system at any time step. Extending this method to any ODE solver simply involves replacing all euler forward solving with whatever ODE solver you prefer.

References

[1] https://arxiv.org/pdf/1806.07366.pdf

[2] https://arxiv.org/pdf/1512.03385.pdf

[3] https://arxiv.org/pdf/1606.04838.pdf

[4] https://arxiv.org/pdf/1502.02551.pdf

[5] https://www.analog.com/media/en/technical-documentation/data-sheets/232512f.pdf