Computing Gradients with Backpropagation

Ryan P. Adams
COS 324 – Elements of Machine Learning
Princeton University

The key insight of neural networks for machine learning is that one can construct powerful (effectively nonparametric) function approximators via the composition of differentiable functions. The backpropagation algorithm is a way to compute the gradients needed to fit the parameters of a neural network, in much the same way we have used gradients for other optimization problems. Backpropagation is a special case of an extraordinarily powerful programming abstraction called *automatic differentiation* (AD). As makes it possible to imperatively write code that computes a function and then have access to the Jacobian of that function for a cost that is only a constant factor worse than the function itself. There are two major flavors of AD: forward mode and reverse mode (although these can be combined in complicated ways). Forward mode AD is a computational abstraction that follows cleanly from our basic intuitions about the chain rule. Reverse mode, however, is less obvious and it requires thinking about *adjoint variables* when computing the chain rule. For the kinds of problems we study in machine learning, reverse mode is almost always what we want, and backpropagation is a particular case of it applied to neural network architectures.

Adaptive Basis Function Regression

Our starting point for neural networks and backprop is to think about least squares regression with basis functions. As before, we imagine we have a set of data $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$ where $\boldsymbol{x}_n \in \mathcal{X}$ and $y_n \in \mathbb{R}$. We define a vector function $\Phi : \mathcal{X} \rightarrow \mathbb{R}^J$ that takes the data and transforms them into a J-dimensional feature vector that we then use in a regression function via:

$$f(\boldsymbol{x}) = \boldsymbol{w}^\mathsf{T}\Phi(\boldsymbol{x}) + b\,. \tag{1}$$

Here I am assuming the bias is separate from the basis. We can use a least squares loss function to measure the quality of any particular $\boldsymbol{w}$ and that allows us to construct an overall objective:

$$\boldsymbol{w}^\star, b^\star = \operatorname*{arg\,min}_{\boldsymbol{w},b} \left\{ \frac{1}{2}\sum_{n=1}^{N}(\boldsymbol{w}^\mathsf{T}\Phi(\boldsymbol{x}_n) + b - y_n)^2 \right\}\,. \tag{2}$$

In this case we can solve for $\boldsymbol{w}$ analytically by solving a linear system, but often we can't do that and we must resort to tools like stochastic gradient descent (SGD). A typical SGD update rule would iterate and in each step take a small subset of the data (or even just a single example), compute the gradient of the loss, and then take a step in the direction of the negative gradient:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \alpha\Phi(\boldsymbol{x}_n)(\boldsymbol{w}^{\mathsf{T}}\Phi(\boldsymbol{x}_n) + b - y_n)\,. \tag{3}$$

Here $\alpha > 0$ is a small constant called the learning rate.

Now, let's imagine that we parameterize the function $\Phi(\cdot)$ with some parameter vector $\boldsymbol{\theta} \in \mathbb{R}^K$, and make this clear by introducing it as a subscript, i.e., $\Phi_{\boldsymbol{\theta}}(\boldsymbol{\theta})$. What these parameters are will depend on the specifics of $\Phi_{\boldsymbol{\theta}}(\cdot)$, but some simple examples for $\mathcal{X} = \mathbb{R}$ and bases we've previously discussed might be:

$$\Phi_{\boldsymbol{\theta}}(x) = \begin{bmatrix} x^{\theta_1} & x^{\theta_2} & \cdots & x^{\theta_K} \end{bmatrix}^{\mathsf{T}} \qquad \text{(polynomial exponents)}$$

$$\Phi_{\boldsymbol{\theta}}(x) = \begin{bmatrix} e^{-(x-\theta_1)^2/\theta_2} & e^{-(x-\theta_3)^2/\theta_4} & \cdots & e^{-(x-\theta_{K-1})^2/\theta_K} \end{bmatrix}^{\mathsf{T}} \qquad \text{(RBF centers and scales)}$$

$$\Phi_{\boldsymbol{\theta}}(x) = \begin{bmatrix} \tanh(x\theta_1 + \theta_2) & \tanh(x\theta_3 + \theta_4) & \cdots & \tanh(x\theta_{K-1} + \theta_K) \end{bmatrix}^{\mathsf{T}} \qquad \text{(tanh scale and location)}$$

Each of these $\Phi_{\boldsymbol{\theta}}(\boldsymbol{x})$ is differentiable with respect to both $\boldsymbol{x}$ and $\boldsymbol{\theta}$. With these new parameters, we can reframe our overall objective function in terms of both $\boldsymbol{w}$ and $\boldsymbol{\theta}$:

$$L(\boldsymbol{w}, b, \boldsymbol{\theta}) = \frac{1}{2}\sum_{n=1}^{N}(\boldsymbol{w}^{\mathsf{T}}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + b - y_n)^2\,. \tag{4}$$

Our minimization problem is the same, just now with $\boldsymbol{\theta}$ in the mix also as above:

$$\boldsymbol{w}^\star, b^\star, \boldsymbol{\theta}^\star = \operatorname*{arg\,min}_{\boldsymbol{w}, b, \boldsymbol{\theta}} \left\{ \frac{1}{2}\sum_{n=1}^{N}(\boldsymbol{w}^{\mathsf{T}}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + b - y_n)^2 \right\}\,. \tag{5}$$

To make gradient updates with respect to $\boldsymbol{\theta}$, we're going to have to differentiate the objective with respect to $\boldsymbol{\theta}$ and that will require the chain rule.

Jacobians and the Chain Rule

We've talked a lot about gradients of scalar functions. However, now we're going to be differentiating vector functions and so now we will need to refresh ourselves on Jacobian matrices. If we have a function $\boldsymbol{f} : \mathbb{R}^K \rightarrow \mathbb{R}^J$, its Jacobian matrix is the matrix of all first derivatives:

$$\mathcal{J}_z\{\boldsymbol{f}(\boldsymbol{z})\} = \begin{bmatrix} \frac{\partial}{\partial z_1} f_1 & \frac{\partial}{\partial z_2} f_1 & \cdots & \frac{\partial}{\partial z_K} f_1 \\ \frac{\partial}{\partial z_1} f_2 & \frac{\partial}{\partial z_2} f_2 & \cdots & \frac{\partial}{\partial z_K} f_2 \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial z_1} f_J & \frac{\partial}{\partial z_2} f_J & \cdots & \frac{\partial}{\partial z_K} f_J \end{bmatrix} \tag{6}$$

In machine learning we tend to think of gradients (arguably incorrectly) as column vectors and so in the case where $J = 1$, the Jacobian is the transpose of the gradient. We could also write the Jacobian as a "stack" of transposed Jacobians:

$$\mathcal{J}_z\{\boldsymbol{f}(z)\} = \begin{bmatrix} (\nabla_z f_1)^\mathsf{T} \\ (\nabla_z f_2)^\mathsf{T} \\ \vdots \\ (\nabla_z f_J)^\mathsf{T} \end{bmatrix} . \tag{7}$$

Neural networks are about composition of differentiable functions, and so the Jacobian is the key object for reasoning about the gradients of these compositions. In particular, if we compose a function $\boldsymbol{g} : \mathbb{R}^J \to \mathbb{R}^M$ the function $\boldsymbol{f}$ above, we get a function $(\boldsymbol{g} \circ \boldsymbol{f}) : \mathbb{R}^K \to \mathbb{R}^M$ with Jacobian:

$$\mathcal{J}\{\boldsymbol{g} \circ \boldsymbol{f}\} = \mathcal{J}\{\boldsymbol{g}\}\mathcal{J}\{\boldsymbol{f}\} . \tag{8}$$

This is the essence of the chain rule for vector functions.

Learning Basis Function Parameters

We now return to the problem of fitting the parameters $\boldsymbol{\theta}$ to data via the optimization problem in Eqn 5. This is a direct use of the chain rule above with the Jacobian of $\Phi_{\boldsymbol{\theta}}(\boldsymbol{x})$ with respect to $\boldsymbol{\theta}$:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \left\{ \frac{1}{2} \sum_{n=1}^{N} (\boldsymbol{w}^\mathsf{T}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + b - y_n)^2 \right\} &= \mathcal{J}_{\boldsymbol{\theta}} \left\{ \frac{1}{2} \sum_{n=1}^{N} (\boldsymbol{w}^\mathsf{T}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + b - y_n)^2 \right\}^\mathsf{T} && (9) \\ &= \frac{1}{2} \sum_{n=1}^{N} \mathcal{J}_{\boldsymbol{\theta}} \left\{ (\boldsymbol{w}^\mathsf{T}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + b - y_n)^2 \right\}^\mathsf{T} && (10) \\ &= \frac{1}{2} \sum_{n=1}^{N} \left(\mathcal{J}_z\{\boldsymbol{w}^\mathsf{T}\boldsymbol{z} + b - y_n)^2\} \mathcal{J}_{\boldsymbol{\theta}}\{\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\} \right)^\mathsf{T} && (11) \\ &= \sum_{n=1}^{N} \left((\boldsymbol{w}^\mathsf{T}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + b - y_n)\boldsymbol{w}^\mathsf{T} \mathcal{J}_{\boldsymbol{\theta}}\{\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\} \right)^\mathsf{T} && (12) \\ &= \sum_{n=1}^{N} \mathcal{J}_{\boldsymbol{\theta}}\{\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\}^\mathsf{T} \boldsymbol{w}(\boldsymbol{w}^\mathsf{T}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + b - y_n) . && (13) \end{aligned}$$

To make it clear that we're looking at a composition with $\Phi_{\boldsymbol{\theta}}(\boldsymbol{x})$, in Eqn 11 above I have used a variable $\boldsymbol{z}$ as the thing we are differentiating with respect to. Just as a sanity check we can see that this has the right dimensional structure. The Jacobian of $\Phi_{\boldsymbol{\theta}}(\boldsymbol{x})$ *with respect to* $\boldsymbol{\theta}$ (not with respect to $\boldsymbol{x}$!) is a $J \times K$ matrix, and $\boldsymbol{w}$ has the dimension of the output of $\Phi_{\boldsymbol{\theta}}(\boldsymbol{x})$, and so is of length J. Therefore the product of the transposed Jacobian and $\boldsymbol{w}$ gives a vector of length K, which is what we want for the gradient of a scalar function with respect to $\boldsymbol{\theta}$.

Having figured out this gradient, we can now write our overall update rules. Here I'm writing them for batch (full-data) gradient descent:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \alpha \sum_{n=1}^{N} \Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n)(\boldsymbol{w}^{\mathsf{T}}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + b - y_n) \tag{14}$$

$$b^{(t+1)} \leftarrow b^{(t)} - \alpha \sum_{n=1}^{N} (\boldsymbol{w}^{\mathsf{T}}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + b - y_n) \tag{15}$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \alpha \sum_{n=1}^{N} \mathcal{J}_{\boldsymbol{\theta}}\{\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\}^{\mathsf{T}}\boldsymbol{w}(\boldsymbol{w}^{\mathsf{T}}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + b - y_n)\,. \tag{16}$$

In practice one might find that different learning rates work better for the different sets of parameters. Note that unlike everything we have looked at so far, this objective is not likely to be convex. Gradient descent may get stuck in various undesirable locations and not find a global minimum.

Automatic Differentiation

The chain rule above told us what we wanted but didn't tell us how to compute it efficiently. Automatic differentiation (AD) is a way to get these gradients efficiently without having to do anything but write the objective function in computer code. AD is a very broadly applicable technique and it has been studied for decades. Curiously however, it has traditionally only been used in a limited way in machine learning despite the ubiquity of gradient-based optimization problems in ML. It has only been recently that serious automatic differentiation (versus naïve hand-coded backprop rules) have started to make their way into mainstream deep learning toolchains. TensorFlow, for example, has the ability to compute gradients of the computational graphs it supports with its limited domain-specific language, but it is not close to a full AD system. Automatic differentiation can be implemented in a variety of ways, via run-time abstractions and also via source code transformation.

Rather than talking about large neural networks, we will seek to understand automatic differentiation via a small problem borrowed from the book of Griewank and Walther (2008). I will use their notation here as well. We wish to compute various derivatives of the function

$$y = [\sin(x_1/x_2) + x_1/x_2 - \exp\{x_2\}] \times [x_1/x_2 - \exp\{x_2\}]\,, \tag{17}$$

evaluated at $x_1 = 1.5$ and $x_2 = 0.5$. If we think about how we might write this to evaluate it in a computer, we might introduce a set of variables that compute pieces and then assemble those pieces together compositionally:

$$\begin{aligned}
v_a &= x_1 & &= 1.5000\\
v_b &= x_2 & &= 0.5000\\
v_1 &= v_a/v_b & = 1.5/0.5 &= 3.0000\\
v_2 &= \sin(v_1) & = \sin(3.0) &= 0.1411\\
v_3 &= \exp\{v_b\} & = \exp\{0.5\} &= 1.6487\\
v_4 &= v_1 - v_3 & = 3.0 - 1.6487 &= 1.3513\\
v_5 &= v_2 + v_4 & = 0.1411 + 1.3513 &= 1.4924\\
v_6 &= v_5 \times v_4 & = 1.4924 \times 1.3513 &= 2.0167\\
y &= v_6 & &= 2.0167
\end{aligned}$$

This is called an *execution trace*.

Forward Mode Automatic Differentiation

The idea of *forward mode* automatic differentiation is that we can compute derivatives as we go and that the chain rule says the overall derivative that we want is a composition of these incremental computations. Let's imagine that our overall goal is to compute $\frac{\partial y}{\partial x_1}$ for the example above. We denote all of the intermediate partial derivatives with respect to x_1 as $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$. Returning to the execution trace, we can get all of the $\dot{v}_i$ just by doing a bit more work at each step.

$$
\begin{aligned}
v_a &= x_1 & &= 1.5000 && (18)\\
\dot{v}_a & & &= 1.0000 && (19)\\
v_b &= x_2 & &= 0.5000 && (20)\\
\dot{v}_b & & &= 0.0000 && (21)\\
v_1 &= v_a/v_b & = 1.5/0.5 &= 3.0000 && (22)\\
\dot{v}_1 &= (v_b\dot{v}_a - v_a\dot{v}_b)/v_b^2 & = (0.5 \times 1.0 - 1.5 \times 0)/0.25^2 &= 2.0000 && (23)\\
v_2 &= \sin(v_1) & = \sin(3.0) &= 0.1411 && (24)\\
\dot{v}_2 &= \cos(v_1) \times \dot{v}_1 & = -0.99 \times 2 &= -1.9800 && (25)\\
v_3 &= \exp\{v_b\} & = \exp\{0.5\} &= 1.6487 && (26)\\
\dot{v}_3 &= v_3 \times \dot{v}_b & = 1.6487 \times 0.0 &= 0.000 && (27)\\
v_4 &= v_1 - v_3 & = 3.0 - 1.6487 &= 1.3513 && (28)\\
\dot{v}_4 &= \dot{v}_1 - \dot{v}_3 & = 2.0 - 0 &= 2.0000 && (29)\\
v_5 &= v_2 + v_4 & = 0.1411 + 1.3513 &= 1.4924 && (30)\\
\dot{v}_5 &= \dot{v}_2 + \dot{v}_4 & = -1.98 + 2.00 &= 0.0200 && (31)\\
v_6 &= v_5 \times v_4 & = 1.4924 \times 1.3513 &= 2.0167 && (32)\\
\dot{v}_6 &= \dot{v}_5 v_4 + \dot{v}_4 v_5 & = 0.02 \times 1.3513 + 1.4924 \times 2.0 &= 3.0118 && (33)\\
y &= v_6 & &= 2.0167 && (34)\\
\dot{y} &= \dot{v}_6 & &= 3.0118 && (35)
\end{aligned}
$$

At the end, we have $\dot{y} = \frac{\partial y}{\partial x_1}$ just by doing some more bookkeeping and computation along the way. The interesting thing is that we can implement this bookkeeping just via abstraction. We can replace our floating point numbers with slightly more complicated objects, but keep things syntactically familiar using *operator overloading*. That is, we can still write Python code like

```
x1 = 1.5
x2 = 0.5
y = (sin(x1/x2) + x1/x2 - exp(x2)) * (x1/x2 - exp(x2))
```

without having to use an awkward domain-specific language embedded within it.

For forward mode we can write something like this:

```
import math

class FwdNumber:
  def __init__(self, value, dotted=0.0):
    '''Create a Forward AD number.  Use dotted to represent its derivative.'''
    self._val = value
    self._dot = dotted

  def __truediv__(self, other):
    '''Floating point division with /'''
    return FwdNumber( self._val / other._val,
                      (self._dot * other._val - other._dot * self._val) / (
                         other._val*other._val))

  def __mul__(self, other):
    '''Floating point multiplication with *'''
    return FwdNumber( self._val * other._val,
                      self._val * other._dot + self._dot * other._val)

  def __add__(self, other):
    '''Floating point addition with +'''
    return FwdNumber( self._val + other._val,
                      self._dot + other._dot)

  def __sub__(self, other):
    '''Floating point subtraction with -'''
    return FwdNumber( self._val - other._val,
                      self._dot - other._dot)

  def __repr__(self):
    '''Print a representation of this object.'''
    return "%f␣%f" % (self._val, self._dot)
```

The `FwdNumber` object is created with two arguments: `val` which is the actual value, and `dotted` which is the derivative. It knows how to interact with other objects of its same type, producing derivatives via the chain rule. We can add a couple of other functions that are friends with the class:

```
def exp(num):
  return FwdNumber( math.exp(num._val),
                    math.exp(num._val) * num._dot)

def sin(num):
  return FwdNumber( math.sin(num._val),
                    math.cos(num._val) * num._dot)
```

Now we can write our own function using these operators:

```
def myfunc(x1, x2):
  a = x1 / x2
  b = exp(x2)
  return (sin(a) + a - b) * (a - b)
```

When we want to compute the function and one of its derivatives, we specify which one we want by making `dotted=1.0` rather than its default of zero:

```
>>> myfunc( FwdNumber(1.5, 1.0), FwdNumber(0.5))
2.016647 3.011843
>>> myfunc( FwdNumber(1.5), FwdNumber(0.5, 1.0))
2.016647 -13.723962
```

We could also add a bit of syntactic sugar via an additional method:

```
  def deriv(self):
    '''Easy way to say that this is the derivative we want.'''
    self._dot = 1.0
    return self
```

Then we can label the derivative:

```
>>> myfunc( FwdNumber(1.5).deriv(), FwdNumber(0.5))
2.016647 3.011843
>>> myfunc( FwdNumber(1.5), FwdNumber(0.5).deriv())
2.016647 -13.723962
```

Note also that `myfunc` can itself be used as a module in other compositions.

Reverse Mode Automatic Differentiation

Reverse mode automatic differentiation proceeds differently. It computes the function, but keeps around information about the structure of the graph and the intermediate variables that were computed. Reverse mode AD then walks backwards through the graph and computes derivatives of the output with respect to the local variable. This quantity is sometimes called an *adjoint variable* and here we denote it as $\bar{v}_i = \frac{\partial y}{\partial v_i}$. This is contrast to forward mode, which computed the derivative of the local variable with respect to one of the inputs, denoted $\dot{i}_i = \frac{\partial v_i}{\partial x_1}$ above. Mathematically, the adjoint is computed by looking at adjoints of the children of vertex v_i on the graph:

$$\bar{v}_i = \sum_{j\text{:child of i}} \bar{v}_j \frac{\partial v_j}{\partial v_i} \tag{36}$$

Since the adjoints depend on the values of their children, we have to go all the way to the end and work backwards:

$$
\begin{aligned}
v_a &= x_1 & &= 1.5000\\
v_b &= x_2 & &= 0.5000\\
v_1 &= v_a/v_b & &= 1.5/0.5 = 3.0000\\
v_2 &= \sin(v_1) & &= \sin(3.0) = 0.1411\\
v_3 &= \exp\{v_b\} & &= \exp\{0.5\} = 1.6487\\
v_4 &= v_1 - v_3 & &= 3.0 - 1.6487 = 1.3513\\
v_5 &= v_2 + v_4 & &= 0.1411 + 1.3513 = 1.4924\\
v_6 &= v_5 \times v_4 & &= 1.4924 \times 1.3513 = 2.0167\\
y &= v_6 & &= 2.0167
\end{aligned}
$$

$$
\begin{aligned}
\bar{v}_6 &= \bar{y} = 1.0 & &\\
\bar{v}_5 &= v_4 \times \bar{v}_6 & &= 1.3513 \times 1.0 = 1.3513\\
\bar{v}_4 &= v_5 \times \bar{v}_6 + \bar{v}_5 & &= 1.4924 \times 1.0 + 1.3513 = 2.8437\\
\bar{v}_3 &= -\bar{v}_4 & &= -2.8437\\
\bar{v}_2 &= \bar{v}_5 & &= 1.3513\\
\bar{v}_1 &= \bar{v}_2 \cos(v_1) + \bar{v}_4 & &= 1.3513 \times -0.99 + 2.8437 = 1.5059\\
\bar{v}_b &= \bar{v}_3 v_3 - \bar{v}_1 v_a/v_b^2 = \bar{v}_3 v_3 - \bar{v}_1 v_1/v_b & &= -2.8437 \times 1.6487 - 1.5059 \times 3/0.5 = -13.7239\\
\bar{v}_a &= \bar{v}_1/v_b & &= 1.5059/0.5 = 3.0118\\
\bar{x}_2 &= \bar{v}_b & &= -13.7239\\
\bar{x}_1 &= \bar{v}_a & &= 3.0118
\end{aligned}
$$

Note that in this case we easily computed *both* $\bar{x}_1 = \frac{\partial y}{\partial x_1}$ and $\bar{x}_2 = \frac{\partial y}{\partial x_2}$, i.e., we computed the gradient $\nabla_{\boldsymbol{x}} y$. This is what backpropagation does. You can see why neural network researchers gave it this name, because the final loss is computing an error and this error is then propagated backwards through the computational graph.

There are various ways to implement this abstraction in its full generality. The three major approaches are:

source code transformation The adjoint backward pass code is generated *a priori* from the forward computation. A clean Python example of such a system is Tangent, at `https://github.com/google/tangent`.

graph-based This approach uses an embedded mini-language to specify a graph of computations that can then be manipulated for function evaluations and gradients. The advantage of this approach is that it is amenable to intelligent graph optimizations and use of compilers. The embedded mini-language also makes it possible to build specialized hardware that targets the differentiable primitives. The downside of this approach is that you are not coding in the host language (e.g., Python) and so you can't take advantage of its imperative design and control flow. Generally the mini-language is less expressive than the host language. Also, the lazy execution of the function represented by the graph can make it difficult to debug. TensorFlow is an example of this kind of automatic differentiation.

tape-based This approach tracks the actual composed functions as they are called during execution of the forward pass. One name for this data structure is the *Wengert list*. With the ordered sequence of computations in hand, it is then possible to walk backward through the list to compute the gradient. The advantage of this is that it can more easily use all the features of the host language and the imperative execution is easier to understand. The downside is that it can be more difficult to optimize the code and reuse computations across executions. Autograd (`https://github.com/HIPS/autograd`) is an example of this. The automatic differentiation in PyTorch (`https://pytorch.org/`) also roughly follows this model.

Reverse Mode Code Example

Here we'll write some very simple code to show how a (naïve) tape-based AD system might work. In forward mode, we replaced our simple numbers with dressed-up tuples and then implemented code to compute both the function and its derivative. In reverse mode, we need more complicated data structures because we need to keep track of the children and capture the execution graph as it is formed. We'll start out by setting up a class with overloaded operators (and some friends):

```
import math

class RevNumber:
  def __init__(self, val):
    self._val      = val
    self._children = []

  def __truediv__(self, other):
    new = RevNumber( self._val / other._val )
    self._children.append( (1.0 / other._val, new) )
    other._children.append( (-self._val/other._val**2, new) )
    return new

  def __mul__(self, other):
    new = RevNumber( self._val * other._val )
    self._children.append( (other._val, new) )
    other._children.append( (self._val, new) )
    return new

  def __add__(self, other):
    new = RevNumber( self._val + other._val )
    self._children.append( (1.0, new) )
    other._children.append( (1.0, new) )
    return new
  def __sub__(self, other):
    new = RevNumber(self._val - other._val )
    self._children.append( (1.0, new) )
    other._children.append( (-1.0, new) )
    return new

def exp(num):
  new = RevNumber( math.exp(num._val) )
  num._children.append( (new._val, new) )
  return new

def sin(num):
  new = RevNumber( math.sin(num._val) )
  num._children.append( (math.cos(num._val), new) )
  return new
```

Here rather than tracking derivatives computed as we go forward, we're tracking the graph itself by storing information about the children of each value. So with an operation like `z3 = z1 + z2`, this would make `z3` a child of both `z1` and `z2`. It is also storing additional information with the parent-to-child edge: the derivative of that child with respect to that parent. This graph is constructed whenever the coded expression executes. We can compose these pieces together just like in forward mode:

```
def myfunc(x1, x2):
  a = x1 / x2
  b = exp(x2)
  return (sin(a) + a - b) * (a - b)
```

Where things get interesting is when we walk the graph backwards via an additional method:

```
  def grad(self, other):
    if self == other:
      return 1.0

    else:
      result = 0.0
      for child in other._children:
        result += child[0] * self.grad(child[1])
      return result
```

You would use this function like this:

```
x1 = RevNumber(1.5)
x2 = RevNumber(0.5)
y  = myfunc(x1, x2)
```

Then you could ask for the function value or also its gradient with respect to something else you have defined:

```
>>> y
2.016647
>>> y.grad(x1)
3.011843327673907
>>> y.grad(x2)
```

```
-13.723961509314076
```

What's happening here is that `grad()` starts by iterating over the children of its argument and then calls `grad()` on each of them, with respect to `self`. It uses the weights it previously stored to then apply the chain rule for itself. This doesn't do any kind of caching or optimization, but is fundamentally what tools like TensorFlow and PyTorch are doing when they compute the gradient of a neural network. This is the backpropagation algorithm.

Changelog

- 21 March 2019 – Expanded to have a reverse-mode code example, and made the forward-mode example use operator overloading.
- 19 October 2018 – Initial version.