Automatic Differentiation

Roger Grosse

Overview

- Implementing backprop by hand is like programming in assembly language.
  - You'll probably never do it, but it's important for having a mental model of how everything works.
- Lecture 6 covered the math of backprop, which you are using to code it up for a particular network for Assignment 1
- This lecture: how to build an automatic differentiation (autodiff) library, so that you never have to write derivatives by hand
  - We'll cover a simplified version of Autograd, a lightweight autodiff tool.
  - PyTorch's autodiff feature is based on very similar principles.

What Autodiff Is Not

- Autodiff is not finite differences.
  - Finite differences are expensive, since you need to do a forward pass for *each* derivative.
  - It also induces huge numerical error.
  - Normally, we only use it for testing.
- Autodiff is both efficient (linear in the cost of computing the value) and numerically stable.

What Autodiff Is Not

- Autodiff is not symbolic differentiation (e.g. Mathematica).
  - Symbolic differentiation can result in complex and redundant expressions.
  - Mathematica's derivatives for one layer of soft ReLU (univariate case):

```
D[Log[1 + Exp[w * x + b]], w]
```

Out[11]=
$$\frac{e^{b+w\,x}\,w}{1+e^{b+w\,x}}$$

  - Derivatives for two layers of soft ReLU:

```
In[19]:= D[Log[1 + Exp[w2 * Log[1 + Exp[w1 * x + b1]] + b2]], w1]
```

Out[19]=
$$\frac{e^{b1+b2+w1\,x+w2\,\mathrm{Log}\left[1+e^{b1+w1\,x}\right]}\,w2\,x}{\left(1+e^{b1+w1\,x}\right)\left(1+e^{b2+w2\,\mathrm{Log}\left[1+e^{b1+w1\,x}\right]}\right)}$$

  - There might not be a convenient formula for the derivatives.
- The goal of autodiff is not a formula, but a procedure for computing derivatives.

What Autodiff Is

An autodiff system should transform the left-hand side into the right-hand side.

Computing the loss:

$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2
\end{aligned}
$$

Computing the derivatives:

$$
\begin{aligned}
\overline{\mathcal{L}} &= 1 \\
\overline{z} &= \overline{y}\,\sigma'(z) \\
\overline{w} &= \overline{z}\,x \\
\overline{b} &= \overline{z}
\end{aligned}
$$

What Autodiff Is

- An autodiff system will convert the program into a sequence of primitive operations which have specified routines for computing derivatives.
- In this representation, backprop can be done in a completely mechanical way.

Original program:

$$z = wx + b$$
$$y = \frac{1}{1 + \exp(-z)}$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Sequence of primitive operations:

$$t_1 = wx$$
$$z = t_1 + b$$
$$t_3 = -z$$
$$t_4 = \exp(t_3)$$
$$t_5 = 1 + t_4$$
$$y = 1/t_5$$
$$t_6 = y - t$$
$$t_7 = t_6^2$$
$$\mathcal{L} = t_7/2$$

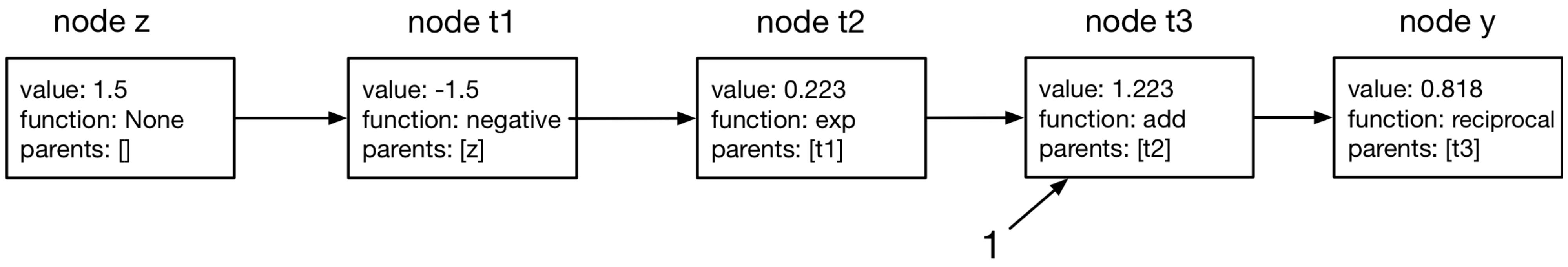
Building the Computation Graph

Example:

```
def logistic(z):
    return 1. / (1. + np.exp(-z))

# that is equivalent to:
def logistic2(z):
    return np.reciprocal(np.add(1, np.exp(np.negative(z))))

z = 1.5
y = logistic(z)
```



Vector-Jacobian Products

- Previously, I suggested deriving backprop equations in terms of sums and indices, and then vectorizing them. But we'd like to implement our primitive operations in vectorized form.
- The Jacobian is the matrix of partial derivatives:

$$\mathbf{J} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

- The backprop equation (single child node) can be written as a vector-Jacobian product (VJP):

$$\overline{x_j} = \sum_i \overline{y_i} \frac{\partial y_i}{\partial x_j} \qquad \qquad \overline{\mathbf{x}} = \overline{\mathbf{y}}^\top \mathbf{J}$$

- That gives a row vector. We can treat it as a column vector by taking

$$\overline{\mathbf{x}} = \mathbf{J}^\top \overline{\mathbf{y}}$$

Vector-Jacobian Products

Examples

- Matrix-vector product

$$\mathbf{z} = \mathbf{W}\mathbf{x} \qquad \mathbf{J} = \mathbf{W} \qquad \overline{\mathbf{x}} = \mathbf{W}^\top \overline{\mathbf{z}}$$

- Elementwise operations

$$\mathbf{y} = \exp(\mathbf{z}) \qquad \mathbf{J} = \begin{pmatrix} \exp(z_1) & & 0 \\ & \ddots & \\ 0 & & \exp(z_D) \end{pmatrix} \qquad \overline{\mathbf{z}} = \exp(\mathbf{z}) \circ \overline{\mathbf{y}}$$

- Note: we never explicitly construct the Jacobian. It's usually simpler and more efficient to compute the VJP directly.

Vector-Jacobian Products

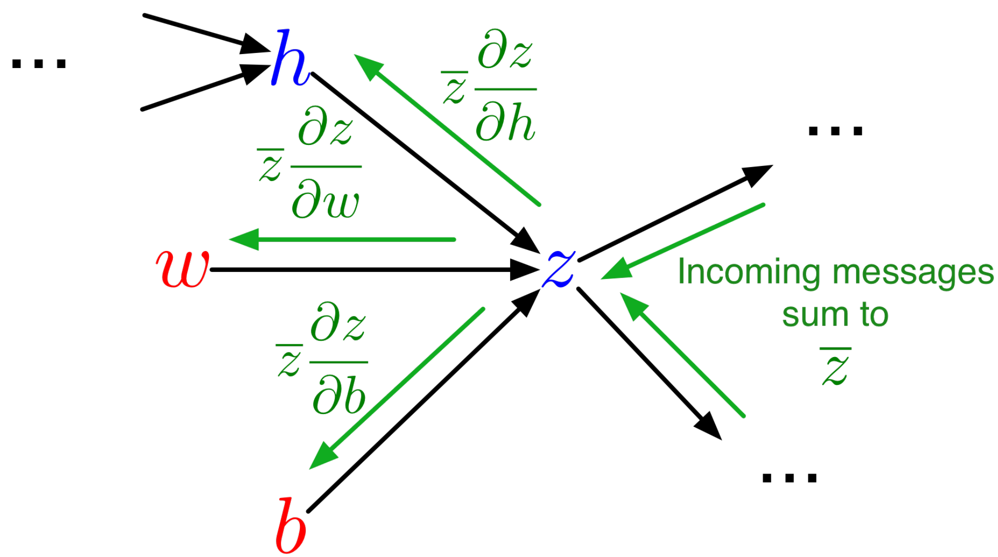
- For each primitive operation, we must specify VJPs for *each* of its arguments. Consider $y = \exp(x)$.
- This is a function which takes in the output gradient (i.e. $\overline{y}$), the answer (y), and the arguments (x), and returns the input gradient ($\overline{x}$)
- `defvjp` (defined in `core.py`) is a convenience routine for registering VJPs. It just adds them to a dict.
- Examples from `numpy/numpy_vjps.py`

```
defvjp(negative, lambda g, ans, x: -g)
defvjp(exp,      lambda g, ans, x: ans * g)
defvjp(log,      lambda g, ans, x: g / x)

defvjp(add,         lambda g, ans, x, y : g,
                    lambda g, ans, x, y : g)
defvjp(multiply,    lambda g, ans, x, y : y * g,
                    lambda g, ans, x, y : x * g)
defvjp(subtract,    lambda g, ans, x, y : g,
                    lambda g, ans, x, y : -g)
```

Backward Pass

- Recall that the backprop computations are more modular if we view them as message passing.



- This procedure can be implemented directly using the data structures we've introduced.

Recap

- We saw three main parts to the code:
  - tracing the forward pass to build the computation graph
  - vector-Jacobian products for primitive ops
  - the backwards pass
- Building the computation graph requires fancy NumPy gymnastics, but other two items are basically what I showed you.

Gradient-Based Hyperparameter Optimization