

CSC321 Lecture 10: 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.

Confusing Terminology

- **Automatic differentiation (autodiff)** refers to a general way of taking a program which computes a value, and automatically constructing a procedure for computing derivatives of that value.
 - In this lecture, we focus on **reverse mode autodiff**. There is also a forward mode, which is for computing directional derivatives.
- **Backpropagation** is the special case of autodiff applied to neural nets
 - But in machine learning, we often use backprop synonymously with autodiff
- **Autograd** is the name of a particular autodiff package.
 - But lots of people, including the PyTorch developers, got confused and started using “autograd” to mean “autodiff”

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):

$$\mathbf{D}[\mathbf{Log}[1 + \mathbf{Exp}[\mathbf{w} * \mathbf{x} + \mathbf{b}]]], \mathbf{w}]$$
$$\text{Out[11]=} \frac{e^{b+w x} w}{1 + e^{b+w x}}$$

- Derivatives for two layers of soft ReLU:

$$\text{In[19]=} \mathbf{D}[\mathbf{Log}[1 + \mathbf{Exp}[\mathbf{w2} * \mathbf{Log}[1 + \mathbf{Exp}[\mathbf{w1} * \mathbf{x} + \mathbf{b1}]]] + \mathbf{b2}]], \mathbf{w1}]$$
$$\text{Out[19]=} \frac{e^{b1+b2+w1 x+w2 \text{Log}[1+e^{b1+w1 x}]} w2 x}{(1 + e^{b1+w1 x}) \left(1 + e^{b2+w2 \text{Log}[1+e^{b1+w1 x}]} \right)}$$

- 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

Recall how we computed the derivatives of logistic least squares regression. An autodiff system should transform the left-hand side into the right-hand side.

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y} = y - t$$

$$\overline{z} = \overline{y} \sigma'(z)$$

$$\overline{w} = \overline{z} x$$

$$\overline{b} = \overline{z}$$

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.

Sequence of primitive operations:

Original program:

$$z = wx + b$$

$$y = \frac{1}{1 + \exp(-z)}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$t_1 = wx$$

$$z = t_1 + b$$

$$t_3 = -z$$

$$t_4 = \exp(t_3)$$

$$t_5 = 1 + t_4$$

$$y = 1/t_5$$

$$t_6 = y - t$$

$$t_7 = t_6^2$$

$$\mathcal{L} = t_7/2$$

What Autodiff Is

```
import autograd.numpy as np
from autograd import grad

def sigmoid(x):
    return 0.5*(np.tanh(x) + 1)

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according to logistic model.
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights):
    # Training loss is the negative log-likelihood of the training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))
```

← very sneaky!

... (load the data) ...

```
# Define a function that returns gradients of training loss using Autograd.
training_gradient_fun = grad(training_loss)

# Optimize weights using gradient descent.
weights = np.array([0.0, 0.0, 0.0])
print "Initial loss:", training_loss(weights)
for i in xrange(100):
    weights -= training_gradient_fun(weights) * 0.01

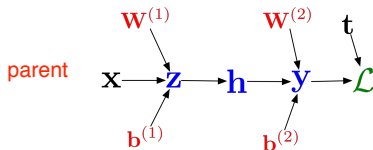
print "Trained loss:", training_loss(weights)
```

← Autograd constructs a
function for computing derivatives
of return value of the function

Autograd

- The rest of this lecture covers how Autograd is implemented.
- Source code for the original Autograd package:
 - <https://github.com/HIPS/autograd>
- Autodidact, a pedagogical implementation of Autograd — you are encouraged to read the code.
 - <https://github.com/mattjj/autodidact>
 - Thanks to Matt Johnson for providing this!

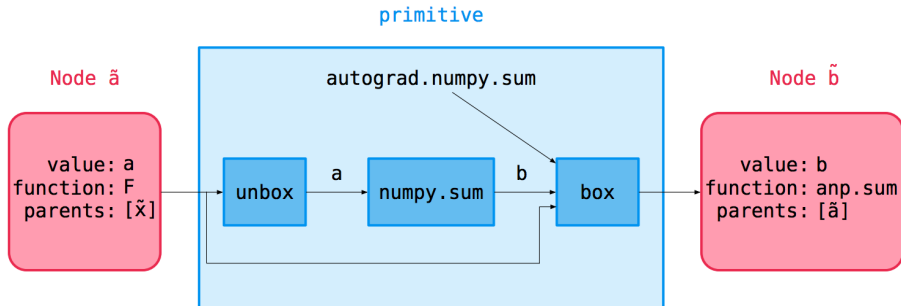
Building the Computation Graph



- Most autodiff systems, including Autograd, explicitly construct the computation graph.
 - Some frameworks like TensorFlow provide mini-languages for building computation graphs directly. Disadvantage: need to learn a totally new API.
 - Autograd instead builds them by **tracing** the forward pass computation, allowing for an interface nearly indistinguishable from NumPy.
- The **Node** class (defined in `tracer.py`) represents a node of the computation graph. It has attributes:
 - `value`, the actual value computed on a particular set of inputs
 - `fun`, the primitive operation defining the node
 - `args` and `kwargs`, the arguments the op was called with
 - `parents`, the parent Nodes **node pass message to parent, so only need pointer to parent**

Building the Computation Graph

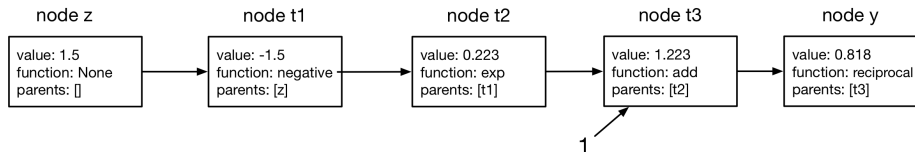
- Autograd's fake NumPy module **provides primitive ops** which look and feel like NumPy functions, but secretly build the computation graph.
- They wrap around NumPy functions:



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)**:

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

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

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

Vector-Jacobian Products

Examples

- Matrix-vector product

$$\mathbf{z} = \mathbf{W}\mathbf{x} \quad \mathbf{J} = \mathbf{W} \quad \bar{\mathbf{x}} = \mathbf{W}^\top \bar{\mathbf{z}}$$

- Elementwise operations

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

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

Vector-Jacobian Products

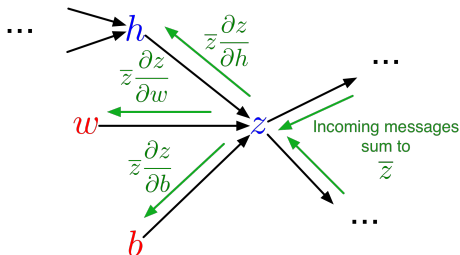
- For each primitive operation, we must specify VJPs for *each* of its arguments. Consider $y = \exp(x)$.
- This is a function which takes in the output gradient (i.e. \bar{y}), the answer (y), and the arguments (x), and returns the input gradient (\bar{x})
- `defvjp` (defined in `core.py`) is a convenience 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.

Backward Pass

- The backwards pass is defined in `core.py`.
- The argument `g` is the error signal for the end node; for us this is always $\bar{\mathcal{L}} = 1$.

```
def backward_pass(g, end_node):
    outgrads = {end_node: g}
    for node in toposort(end_node):
        outgrad = outgrads.pop(node)
        fun, value, args, kwargs, argnums = node.recipe
        for argnum, parent in zip(argnums, node.parents):
            vjp = primitive_vjps[fun][argnum]
            parent_grad = vjp(outgrad, value, *args, **kwargs)
            outgrads[parent] = add_outgrads(outgrads.get(parent), parent_grad)
    return outgrad

def add_outgrads(prev_g, g):
    if prev_g is None:
        return g
    return prev_g + g
```

Backward Pass

- `grad` (in `differential_operators.py`) is just a wrapper around `make_vjp` (in `core.py`) which builds the computation graph and feeds it to `backward_pass`.
- `grad` itself is viewed as a VJP, if we treat $\bar{\mathcal{L}}$ as the 1×1 matrix with entry 1.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial \mathbf{w}} \bar{\mathcal{L}}$$

```
def make_vjp(fun, x):
    """Trace the computation to build the computation graph, and return
    a function which implements the backward pass."""
    start_node = Node.new_root()
    end_value, end_node = trace(start_node, fun, x)
    def vjp(g):
        return backward_pass(g, end_node)
    return vjp, end_value

def grad(fun, argnum=0):
    def gradfun(*args, **kwargs):
        unary_fun = lambda x: fun(*subval(args, argnum, x), **kwargs)
        vjp, ans = make_vjp(unary_fun, args[argnum])
        return vjp(np.ones_like(ans))
    return gradfun
```

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.
- You're encouraged to read the full code (< 200 lines!) at:

<https://github.com/mattjj/autodidact/tree/master/autograd>