CSC421/2516 Lecture 6: Automatic Differentiation

Roger Grosse and Jimmy Ba
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 4 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: Finite Differences

- We often use finite differences to check our gradient calculations.
  - One-sided version:
    \[
    \frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i, \ldots, x_N)}{h}
    \]
  - Two-sided version:
    \[
    \frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i - h, \ldots, x_N)}{2h}
    \]
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: Symbolic Differentiation

- 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):
    \[
    \frac{e^{b+wx}w}{1+e^{b+wx}}
    \]
  - Derivatives for two layers of soft ReLU:
    \[
    \frac{e^{b1+b2+w1x+w2\log[1+e^{b1+w1x}]}w2x}{(1+e^{b1+w1x})(1+e^{b2+w2\log[1+e^{b1+w1x}]})}
    \]
- There might not be a convenient formula for the derivatives.
- The goal of autodiff is not a formula, but a procedure for computing derivatives.
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:**

\[ \bar{\mathcal{L}} = 1 \]
\[ \bar{y} = y - t \]
\[ \bar{z} = \bar{y} \sigma'(z) \]
\[ \bar{w} = \bar{z} x \]
\[ \bar{b} = \bar{z} \]
What Autodiff Is

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

Sequence of primitive operations:

\[
\begin{align*}
t_1 &= wx \\
z &= t_1 + b \\
t_3 &= -z \\
t_4 &= \exp(t_3) \\
t_5 &= 1 + t_4 \\
y &= 1/t_5 \\
t_6 &= y - t \\
t_7 &= t_6^2 \\
\mathcal{L} &= t_7/2
\end{align*}
\]

Original program:

\[
\begin{align*}
z &= wx + b \\
y &= \frac{1}{1 + \exp(-z)} \\
\mathcal{L} &= \frac{1}{2}(y - t)^2
\end{align*}
\]
What Autodiff Is

```python
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))

# 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)
```

... (load the data) ...

**very sneaky!**

Autograd constructs a function for computing derivatives
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!
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
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:

```python
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)
```
Recap: Vector-Jacobian Products

- Recall: the **Jacobian** is the matrix of partial derivatives:

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

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

  \[
  \bar{x}_j = \sum_i y_i \frac{\partial y_i}{\partial x_j} \quad \bar{x} = \bar{y}^\top J
  \]

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

  \[
  \bar{x} = J^\top \bar{y}
  \]
Recap: Vector-Jacobian Products

Examples

- **Matrix-vector product**
  
  \[ z = Wx \quad J = W \quad \overline{x} = W^\top \overline{z} \]

- **Elementwise operations**
  
  \[
  y = \exp(z) \quad J = \begin{pmatrix}
  \exp(z_1) & 0 & \cdots & 0 \\
  0 & \ddots & \ddots & 0 \\
  0 & \cdots & 0 & \exp(z_D)
  \end{pmatrix} \quad \overline{z} = \exp(z) \circ \overline{y}
  \]

- **Note**: we never explicitly construct the Jacobian. It’s usually simpler and more efficient to compute the VJP directly.
Backprop as Message Passing

Consider a naïve backprop implementation where the \( z \) module needs to compute \( \bar{z} \) using the formula:

\[
\bar{z} = \frac{\partial r}{\partial z} \bar{r} + \frac{\partial s}{\partial z} \bar{s} + \frac{\partial t}{\partial z} \bar{t}
\]

This breaks modularity, since \( z \) needs to know how it’s used in the network in order to compute partial derivatives of \( r \), \( s \), and \( t \).
Backprop as message passing:

- Each node receives a bunch of messages from its children, which it aggregates to get its error signal. It then passes messages to its parents.
- Each of these messages is a VJP.
- This formulation provides modularity: each node needs to know how to compute its outgoing messages, i.e. the VJPs corresponding to each of its parents (arguments to the function).
- The implementation of $z$ doesn’t need to know where $\bar{z}$ came from.
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`

```python
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

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

```python
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 $\overline{L}$ as the $1 \times 1$ matrix with entry 1.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial w} \overline{L}$$

```python
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
Differentiating through a Fluid Simulation

```python
def project(vx, vy):
    # Project the velocity field to be approximately mass-conserving,
    # using a few iterations of Gauss-Seidel.
    p = np.zeros(vx.shape)
    h = 1.0/vx.shape[0]
    div = -0.5 * h * (np.roll(vx, -1, axis=0) - np.roll(vx, 1, axis=0)
                     + np.roll(vy, -1, axis=1) - np.roll(vy, 1, axis=1))
    for k in range(10):
        p = (div + np.roll(p, 1, axis=0) + np.roll(p, -1, axis=0)
             + np.roll(p, 1, axis=1) + np.roll(p, -1, axis=1))/4.0
    vx -= 0.5*(np.roll(p, -1, axis=0) - np.roll(p, 1, axis=0))/h
    vy -= 0.5*(np.roll(p, -1, axis=1) - np.roll(p, 1, axis=1))/h
    return vx, vy

def advect(f, vx, vy):
    # Move field f according to x and y velocities (u and v)
    # using an implicit Euler integrator.
    rows, cols = f.shape
    cell_xs, cell_ys = np.meshgrid(np.arange(rows),
                                    np.arange(cols))
    center_xs = (cell_xs - vx).ravel()
    center_ys = (cell_ys - vy).ravel()

    # Compute indices of source cells.
    left_ix = np.floor(center_xs).astype(int)
    top_ix = np.floor(center_ys).astype(int)
    rw = center_xs - left_ix
    bw = center_ys - top_ix
    left_ix = np.mod(left_ix, rows)
    right_ix = np.mod(left_ix + 1, rows)
    top_ix = np.mod(top_ix, cols)
    bot_ix = np.mod(top_ix + 1, cols)

    flat_f = (1 - rw) * (((1 - bw)*f[left_ix, top_ix] \
                          + bw*f[left_ix, bot_ix]) / \
                        + rw * (((1 - bw)*f[right_ix, top_ix] \
                               + bw*f[right_ix, bot_ix]))

    return np.reshape(flat_f, (rows, cols))

def simulate(vx, vy, smoke, num_time_steps):
    for t in range(num_time_steps):
        vx_updated = advect(vx, vx, vy)
        vy_updated = advect(vy, vx, vy)
        vx, vy = project(vx_updated, vy_updated)
        smoke = advect(smoke, vx, vy)
    return smoke, frame_list
```

Roger Grosse and Jimmy Ba
CSC421/2516 Lecture 6: Automatic Differer
Differentiating through a Fluid Simulation

https://github.com/HIPS/autograd#end-to-end-examples
Gradient-Based Hyperparameter Optimization

regularization params

optimization params

training data

validation data

gradient descent

init Θ

θ

loss grad

update step

∇L

θ

loss grad

update step

∇L

θ_{final}

validation set error

L
Gradient-Based Hyperparameter Optimization

P(digit | image)

![Graph showing learning rate vs schedule index for different layers: Layer 1, Layer 2, Layer 3, Layer 4. The graph illustrates how the learning rate changes over the schedule index.](image)