AUTOGRAD TUTORIAL

Paul Vicol, Slides Based on Ryan Adams'

January 30, 2017

CSC 321, University of Toronto

1. Automatic Differentiation
2. Introduction to Autograd
3. IPython Notebook Demo

To solve a problem using machine learning you generally need to:

1. Define a *model* f_θ governed by parameters θ
2. Come up with a *loss function* $\mathcal{L}$ that quantifies how well your model fits the data
3. *Optimize* the loss function with respect to the parameters θ

- To optimize $\mathcal{L}$ w.r.t θ, we need to find the *gradient* $\nabla_\theta \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \theta}$

- **Symbolic differentiation:** Automatic manipulation of mathematical expressions to get derivatives
  - Input and output are mathematical expressions
  - Used in Mathematica, Maple, Sympy, etc.
- **Numeric differentiation:** Approximating derivatives by finite differences:

$$\frac{\partial}{\partial x_i} f(x_1, \dots, x_N) = \lim_{h \to 0} \frac{f(x_1, \dots, x_i + h, \dots, x_N) - f(x_1, \dots, x_i - h, \dots, x_N)}{2h}$$

- **Automatic differentiation (AD):** A method to get exact derivatives efficiently, by storing information as you go forward that you can reuse as you go backwards
  - Takes code that computes a function and returns code that computes the derivative of that function.
  - "The goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives."
  - Autograd, Torch Autograd

- Automatic differentiation is a set of abstractions that enable you to write a function and efficiently apply the chain rule to it

Main Idea:

1. All numeric computations are compositions of a finite set of elementary operations (`+`, `-`, `*`, `/`, `exp`, `log`, `sin`, `cos`, etc.)
2. We can write code to differentiate these basic operations
3. When we encounter a complicated function we break it down and deal with those basic ops as opposed to finding the gradient of the entire computation.

- **Autograd** is a Python package for automatic differentiation
- To install Autograd:

  ```
  pip install autograd
  ```

- Autograd can automatically differentiate Python and Numpy code
- It can handle most of Python's features, including loops, if statements, recursion and closures
- It can also compute higher-order derivatives
- Uses reverse-mode differentiation (backpropagation) so it can efficiently take gradients of scalar-valued functions with respect to array-valued or vector-valued arguments.

```python
# Thinly wrapped numpy
import autograd.numpy as np

# Basically everything you need
from autograd import grad

# Define a function like normal with Python and Numpy
def tanh(x):
    y = np.exp(-x)
    return (1.0 - y) / (1.0 + y)

# Create a function to compute the gradient
grad_tanh = grad(tanh)

# Evaluate the gradient at x = 1.0
print(grad_tanh(1.0))
```

```
# Taylor approximation to sin function
def fun(x):
    currterm = x
    ans = currterm
    for i in range(1000):
        print(i, end=' ')
        currterm = - currterm * x ** 2 /
                   ((2 * i + 3) * (2 * i + 2))
        ans = ans + currterm
        if np.abs(currterm) < 0.2:
            break

    return ans

d_fun = grad(fun)
dd_fun = grad(d_fun) # Second-order gradient
```

- Autograd allows you to compute gradients of many types of data structures
    - Any nested combination of lists, tuples, arrays, or dicts
- The `flatten` function converts data structures to *1-D vectors*
    - We know how to compute gradients of vectors
    - To compute gradients of more complicated structures, convert the structures to vectors, perform computations, and then convert back to the original data structure
- Provides a lot of flexibility in how you store and manipulate the parameters of your model

There are several reasons you might want to do this, including:

1. **Speed:** You may know a faster way to compute the gradient for a specific function.
2. **Numerical Stability**
3. When your code depends on **external library calls**

```
from autograd import primitive
@primitive
def logsumexp(x):
    return ...

# Define a custom gradient function
def make_grad_logsumexp(ans, x):
    def gradient_product(g):
        return ...
    return gradient_product

# Tell autograd about the custom gradient function
logsumexp.defgrad(make_grad_logsumexp)
```

- Two approaches to automatic differentiation: **explicit** vs **implicit** computational graph construction.
- Various tools implement limited forms of automatic differentiation using mini-languages
- Many deep learning packages involve *explicit graph construction*, including:
  - Theano
  - Caffe
  - Vanilla Torch (as compared to Autograd for Torch)
  - Tensorflow
- On the other hand, Autograd implicitly constructs a computational graph by tracking operations
- Review paper: Baydin, Pearlmutter, Radul & Siskind "Automatic Differentiation in Machine Learning: A Survey" `http://arxiv.org/abs/1502.05767`

IPYTHON NOTEBOOK EXAMPLE