Lecture 6: Automatic Differentiation

Roger Grosse

1 Introduction

Last week, we saw how the backpropagation algorithm could be used to compute gradients for basically any neural net architecture, as long as it's a feed-forward computation and all the individual pieces of the computation are differentiable. Backprop is based on the computation graph, and it basically works backwards through the graph, applying the chain rule at each node. In that lecture, we would derive the algorithm by hand, in a form that could be translated into a NumPy program. This is how one would have implemented a neural net 10 years ago.

Implementing backprop manually is like writing assembly language: it's good to do a couple times so that you understand how things work beneath the hood.

But, as you may have noticed, the procedure we developed was entirely mechanical. In this lecture, we'll see how to write an *automatic differentiation engine* — a program that builds the computation graph and applies the backprop updates, all without us having to calculate any derivatives by hand. Over the past decade, automatic differentiation frameworks such as Theano, Autograd, TensorFlow, and PyTorch have made it incomparably easier to implement backprop for fancy neural net architectures, thereby dramatically expanding the range and complexity of network architectures we're able to train.

Autodiff has existed since the 70s, but somehow ML people never picked it up until the past decade, despite spending countless hours calculating derivatives.

In this lecture, we'll focus on Autograd[1], a lightweight automatic differentiation system written by Dougal Maclaurin, David Duvenaud, Matt Johnson, and Jamie Townsend. While the major neural net frameworks (TensorFlow, PyTorch, etc.) have giant codebases, much of their complexity comes from supporting a wide variety of neural net layers and operations, and from making sure everything gets the maximum performance out of the GPU. By contrast, Autograd is just an autodiff package: it doesn't include GPU support, and it's not aiming for comprehensive coverage of modern neral net architectures. But this means the implementation is very clean and simple. We'll actually be using a stripped-down, pedagogical implementation of Autograd called Autodidact[2], whose core funcationality is implemented in less than 200 lines of Python code!

Currently, Autograd isn't used much for production neural nets due to its lack of GPU support. For the remainder of the course, we'll use PyTorch, a more comprehensive neural net framework whose autodiff functionality is loosely based on Autograd. But if you wish you could just use Autograd for everything, you're in luck: some of the Autograd creators are working on a framework called JAX[3] which compiles Autograd computation graphs into efficient GPU code. It's a bit experimental now (having been released

[1] `https://github.com/HIPS/autograd`

[2] `https://github.com/mattjj/autodidact`

[3] `https://github.com/google/jax`

December 2018), but you might see this gain more traction over the next few years because of its slick, NumPy-like API.

Some clarifications about terminology are in order. In our field, we usually use "backprop" to refer to the mathematical algorithm and "autodiff" to refer to a software implementation, but the terms are somewhat interchangeable. Sometimes "backprop" just refers to autodiff applied to neural nets, though really the algorithm is no different from the more general case. "Autograd" is the name of a particular software package, but it's often used incorrectly as a generic term for autodiff (e.g. "PyTorch Autograd"). The particular kind of autodiff we use to compute gradients is known as **reverse mode autodiff** because it goes backwards through the computation graph. There's also a **forward mode autodiff**, which is used to compute directional derivatives. We won't used forward mode in this class.

In one of the most elegant pieces of code I've ever seen, it's possible to implement forward mode autodiff in only three lines by calling reverse mode twice. https://github.com/renmengye/tensorflow-forward-ad/issues/2

2 What Autodiff Isn't

One way to motivate autodiff is to contrast it with some related but distinct concepts.

2.1 Autodiff is not finite differences

An easy way to compute derivatives is to approximate them numerically using **finite differences**, or **numerical differentiation**. We just use the definition of derivatives in terms of a limit, but simply plug in a small value of h. The most obvious is the one-sided definition:

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i, \ldots, x_N)}{h}$$

There's also a more numerically stable two-sided version:

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i - h, \ldots, x_N)}{2h}$$

The approximations are illustrated in Figure 1. Finite differences are commonly used to test the implementation of gradient computations. I.e., we check that the analytic form for the derivatives is close to the finite difference approximation. Since finite differences only requires calculating function values, this is a pretty easy test to write.

However, finite differences is not a practical way to compute derivatives for neural net training. The most obvious reason is that it's very expensive: you need to do a separate forward pass for *each* partial derivative. It's also numerically unstable, since you first subtract two very close values and then divide by a small number. By contrast, autodiff is both efficient and numerically stable.

2.2 Autodiff is not symbolic differentiation

If you've used a mathematical computing environment such as Mathematica or Maple, it's likely you've made use of **symbolic differentiation**. Here, the aim is to take a mathematical expression and return a mathematical expression for the derivative. This is convenient for relatively simple expressions:

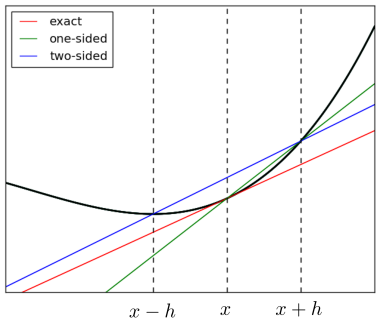


Figure 1: One-sided vs. two-sided finite differences.

```
In[11]:= D[Log[1 + Exp[w * x + b]], w]
Out[11]= e^(b+w x) w / (1 + e^(b+w x))
```

but can get unwieldy for more complex expressions:

```
In[19]:= D[Log[1 + Exp[w2 * Log[1 + Exp[w1 * x + b1]] + b2]], w1]
Out[19]= e^(b1+b2+w1 x+w2 Log[1+e^(b1+w1 x)]) w2 x / ((1 + e^(b1+w1 x)) (1 + e^(b2+w2 Log[1+e^(b1+w1 x)])))
```

This is unavoidable in symbolic differentiation, since there might not *be* a convenient expression for the derivative.

But notice that part of the problem here is that the expression has a bunch of repeated terms. Imagine we instead defined variables to represent some of these terms, e.g. `z = b1 + w1 x`, and simply referenced those variables in the expression. If you do this consistently enough, you'll basically wind up with autodiff.

3 What autodiff is

Recall from Lecture 4, when we came up with a procedure to compute the derivatives of a simple univariate cost function:

Computing the loss:

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

Computing the derivatives:

$$\overline{\mathcal{L}} = 1$$
$$\overline{y} = y - t$$
$$\overline{z} = \overline{y}\,\sigma'(z)$$
$$\overline{w} = \overline{z}\,x$$
$$\overline{b} = \overline{z}$$

Previously, we would implement a procedure like this as a Python program. But an autodiff package would build up data structures to represent these computations, and then it can simply execute the right-hand side.

One thing we'll change from Lecture 4 is the level of granularity. When we derived backprop by hand, we drew a computation graph in terms of high-level variables that were meaningful in terms of the neural net architecture, and each node corresponded to a mathematical expression that might involve multiple operations. But if the computer is going to do all the work, we might as well define a more fine-grained computation graph, where the nodes correspond to individual **primitive operations**, or **ops**. Ops are basically simple operations, such as multiplication, for which we directly implement the derivatives. An autodiff package would typically break the above computation down into a sequence of primitive ops:

Original program:

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

Sequence of ops:

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

This is all invisible to the user. To the user, it feels like you just write ordinary code for the forward pass, and then call a function to compute the derivatives. E.g., Figure 2 shows an implementation of gradient descent for logistic regression in Autograd. This program includes a call to `grad`, which takes in a function and spits out another function which computes its derivatives. Essentially, this one function is the *only* API you need to learn to use Autograd. Other than that, writing Autograd code looks and feels like writing NumPy. Autograd achieves this by defining its own NumPy package (`autograd.numpy`), which exposes roughly the same API as NumPy, but which does a bunch of additional bookkeeping to build the computation graph needed by `grad`.

4 Implementing autodiff

We now see how to implement a simple but powerful autodiff system. We'll follow the implementation of Autodidact[4], a simpler pedagogical implementation of Autograd. But despite its simplicity, it still gives the full power of autodiff; for instance, you can easily differentiate through a backprop computation (i.e. compute second derivatives by calling `grad` twice), something which is actually fairly awkward to do in TensorFlow or PyTorch! You're encouraged to read the code; the core functionality is less than 200 lines of Python!

There are basically three parts to the Autodidact implementation:

1. Tracing the computation to build the computation graph

[4] `https://github.com/mattjj/autodidact`

```
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

Figure 2: Logistic regression implemented in Autograd.

2. Implementing vector-Jacobian products for each primitive op
3. Backprop itself

4.1 Tracing the computation

Since backprop is done on the computation graph, any autodiff package must first somehow build the computation graph. The approach taken by TensorFlow is for the user to build the graph directly. I.e., the user explicitly builds the graph node by node, and then executes it. PyTorch and Autograd instead build the graph implicitly by **tracing** the computation in the forward pass. This results in a much cleaner interface, because the user doesn't have to worry about any distinction between graph building and execution phases.

The main building block of the computation graph is the `Node` class, which (as you might guess) represents one node of the graph. It keeps track of four pieces of information:

1. `value`, the actual value computed on a particular set of inputs
2. `fun`, the primitive operation defining the node
3. `args` and `kwargs`, the arguments the op was called with
4. `parents`, the parent `Nodes`

In order to implicitly build the computation graph, the `autograd.numpy` module defines an API which looks and feels like NumPy, but where each op

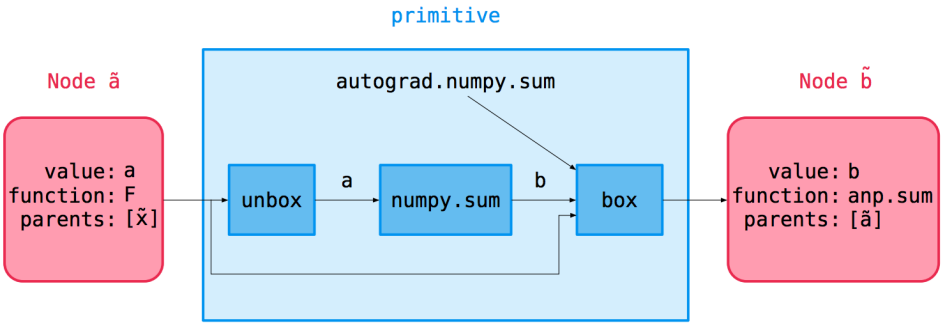


Figure 3: `autograd.numpy` wraps around NumPy operations to implicitly build the computation graph.

```
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)
```

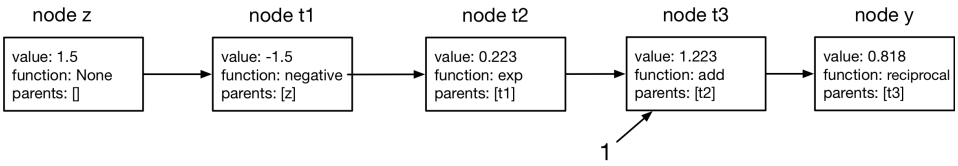


Figure 4: Computation graph built for a simple program. The function `logistic2` is simply an explicit representation of the NumPy functions called when you use arithmetic operators.

wraps around the corresponding NumPy function and, instead of returning an array, returns a `Node` instance containing the array. In particular, it first **unboxes** its inputs (retrieves the values from the `Node` objects), feeds those values to the NumPy function, and then **boxes** the result into a `Node` instance, as shown in Figure 3. Figure 4 shows an example of a small computation graph built up in this way.

The code that builds the computation graph requires the fanciest Python gymnastics of the whole package. We're not going to look at it in detail, but I encourage you to read it.

4.2 Vector-Jacobian products

Recall from Lecture 4 that the vectorized version of Backprop is defined in terms of **vector-Jacobian products (VJPs)**. The **Jacobian matrix** is

the matrix of partial derivatives:

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

Recall that the error signal for each node is computed using the formula

$$\overline{\mathbf{v}_i} = \sum_{j \in \mathrm{Ch}(\mathbf{v}_i)} \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i}^\top \overline{\mathbf{v}_j}.$$

This can be equivalently written as

$$\overline{\mathbf{v}_i}^\top = \sum_{j \in \mathrm{Ch}(\mathbf{v}_i)} \overline{\mathbf{v}_j}^\top \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i},$$

emphasizing that each piece of the computation involves multiplying a vector by the Jacobian. Hence the term VJP.

Note that we *do not* explicitly construct the Jacobian in order to compute a VJP. For instance, if a node represents a simple elementwise operation, e.g.

$$\mathbf{y} = \exp(\mathbf{z}),$$

then the Jacobian is a diagonal matrix:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \exp(z_1) & & 0 \\ & \ddots & \\ 0 & & \exp(z_D) \end{pmatrix}.$$

Think about why the Jacobian is diagonal. Why isn't this true of, say, softmax? How would you efficiently implement a VJP for softmax?

This matrix is size $D \times D$, and the explicit matrix-vector product would require $\mathcal{O}(D^2)$ operations. But the VJP itself can be implemented in linear time:

$$\overline{\mathbf{z}} = \frac{\partial \mathbf{y}}{\partial \mathbf{z}}^\top \overline{\mathbf{y}} = \exp(\mathbf{z}) \circ \overline{\mathbf{y}}.$$

Hence, we need to write procedures to compute VJPs, but we never actually construct the Jacobian.

Take a look at `numpy/numpy_vjps.py`. This module defines VJPs for each NumPy op. Each line is a call to `defvjp` which is a thin wrapper which just adds stuff to a Python dict which stores all the VJP functions. For each op, we need to specify a VJP for *each* of its arguments. The VJP is represented as a function which takes in the output error signal `g`, the value of the node `ans`, and the arguments to the op (which, remember, are `Node` instances). The function returns the input error signal for the corresponding argument. Some examples are shown in Figure 5

Exercise: write VJPs for division and elementwise log.

4.3 Backprop

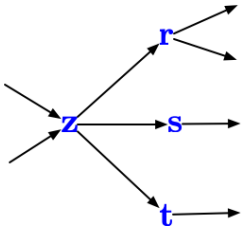
Now we can finally implement backprop! Tracing the computation required the fanciest Python tricks, and implementing VJPs required the most lines of code. It turns out that backprop itself is pretty straightforward.

The one conceptual leap is to think about it as a message-passing procedure. Consider the following computation graph:

```
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)
```

Figure 5: Some example VJPs defined in `numpy/numpy_vjps.py`.

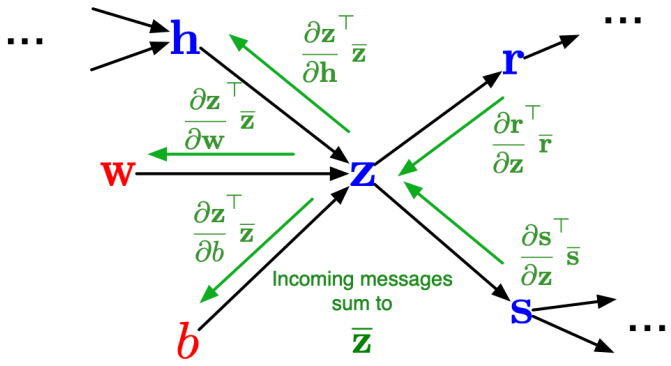


The way we described it in Lecture 4, in order to compute $\bar{\mathbf{z}}$, you'd find the expressions for its child nodes, and differentiate each of them weigh respect to $\mathbf{z}$:

$$\bar{\mathbf{z}} = \frac{\partial \mathbf{r}}{\partial \mathbf{z}}\bar{\mathbf{r}} + \frac{\partial \mathbf{s}}{\partial \mathbf{z}}\bar{\mathbf{s}} + \frac{\partial \mathbf{t}}{\partial \mathbf{z}}\bar{\mathbf{t}}$$

The problem is, this breaks modularity because now the implementation of $\mathbf{z}$ needs to understand the implementations of all its child nodes in order to compute its error signal.

Instead, we can reformulate the algorithm in terms of messages being passed between nodes. Consider the following portion of a graph:



$\mathbf{z}$ receives messages from each of its children, corresponding to their VJPs. But $\mathbf{z}$ doesn't know the messages correspond to VJPs; all it knows is that it needs to sum them together to compute $\bar{\mathbf{z}}$. Once it does that, it computes the VJPs with respect to each of its arguments, and sends them as messages to each of its parent nodes. If you think about it, this is exactly the same sequence of computations as we discussed in Lecture 4, but now each node only has to understand how to compute its own VJPs.

Finally, with this understanding, here's the code which does backprop. It takes two arguments: `end_node`, the output node of the computation graph, and `g`, the output error signal $\overline{\mathcal{L}}$. In this course, we never have any reason to use anything other than the scalar value 1 for the output error signal. However, Autograd lets you use vector-valued outputs, and you can specify any error signal you like.

One use for more general output error signals is if you want to hardcode the backprop procedure for a network while still using autodiff to backprop through individual pieces of it. One example of this is RevNets (Gomez et al., "The reversible residual network: backprop without storing activations"), which uses a special backprop procedure that avoids storing the activations in memory.

```
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
```

The `grad` function is simply a shallow wrapper around `backward_pass`. We haven't talked about it this way, but the entire backprop algorithm is really just computing a VJP, where the Jacobian is of the entire forward computation (viewed as a function mapping parameters to loss). Here's the code:

```
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
```

So, that's it. That describes the whole implementation of the core functionality of an autodiff package. You're encouraged to read the Autodidact code[5]. Also, definitely check out the Autograd examples page[6], which contains lots of fun examples like computing higher-order derivatives and differentiating through a fluid dynamics simulator.

[5] `https://github.com/mattjj/autodidact`
[6] `https://github.com/HIPS/autograd`