Appendix D – Autodiff

This notebook contains toy implementations of various autodiff techniques, to explain how they work.

Open in Colab

Setup

Introduction

Suppose we want to compute the gradients of the function $f(x,y) = x^2y + y + 2$ **with regards to the parameters x and y:**

In [1]:

```python
def f(x,y):
    return x*x*y + y + 2
```

One approach is to solve this analytically:

$$\frac{\partial f}{\partial x} = 2xy$$

$$\frac{\partial f}{\partial y} = x^2 + 1$$

In [2]:

```python
def df(x,y):
    return 2*x*y, x*x + 1
```

So for example $\frac{\partial f}{\partial x}(3, 4) = 24$ **and** $\frac{\partial f}{\partial y}(3, 4) = 10$.

In [3]:

```python
df(3, 4)
```

Out[3]:

```
(24, 10)
```

Perfect! We can also find the equations for the second order derivatives (also called Hessians):

$$\frac{\partial^2 f}{\partial x \partial x} = \frac{\partial(2xy)}{\partial x} = 2y$$

$$\frac{\partial^2 f}{\partial x \partial y}$$

$\partial x \partial y$

$= \dfrac{\partial(2xy)}{\partial y}$

$= 2x$

$\dfrac{\partial^2 f}{\partial y \partial x}$

$= \dfrac{\partial(x^2 + 1)}{\partial x}$

$= 2x$

$\dfrac{\partial^2 f}{\partial y \partial y}$

$= \dfrac{\partial(x^2 + 1)}{\partial y}$

$= 0$

At x=3 and y=4, these Hessians are respectively 8, 6, 6, 0. Let's use the equations above to compute them:

In [4]:

```
def d2f(x, y):
    return [2*y, 2*x], [2*x, 0]
```

In [5]:

```
d2f(3, 4)
```

Out[5]:

```
([8, 6], [6, 0])
```

Perfect, but this requires some mathematical work. It is not too hard in this case, but for a deep neural network, it is pratically impossible to compute the derivatives this way. So let's look at various ways to automate this!

Numeric differentiation

Here, we compute an approxiation of the gradients using the equation: $\dfrac{\partial f}{\partial x} = \lim\limits_{\epsilon \to 0} \dfrac{f(x+\epsilon, y) - f(x, y)}{\epsilon}$ **(and there is a similar definition for** $\dfrac{\partial f}{\partial y}$**).**

In [6]:

```
def gradients(func, vars_list, eps=0.0001):
    partial_derivatives = []
    base_func_eval = func(*vars_list)
    for idx in range(len(vars_list)):
        tweaked_vars = vars_list[:]
        tweaked_vars[idx] += eps
        tweaked_func_eval = func(*tweaked_vars)
        derivative = (tweaked_func_eval - base_func_eval) / eps
        partial_derivatives.append(derivative)
    return partial_derivatives
```

In [7]:

```
def df(x, y):
    return gradients(f, [x, y])
```

In [8]:

```
df(3, 4)
```

Out[8]:

```
[24.000400000048216, 10.000000000047748]
```

It works well!

The good news is that it is pretty easy to compute the Hessians. First let's create functions that compute the first order partial derivatives (also called Jacobians):

In [9]:

```
def dfdx(x, y):
    return gradients(f, [x,y])[0]

def dfdy(x, y):
    return gradients(f, [x,y])[1]

dfdx(3., 4.), dfdy(3., 4.)
```

Out[9]:

```
(24.000400000048216, 10.000000000047748)
```

Now we can simply apply the `gradients()` function to these functions:

In [10]:

```
def d2f(x, y):
    return [gradients(dfdx, [x, y]), gradients(dfdy, [x, y])]
```

In [11]:

```
d2f(3, 4)
```

Out[11]:

```
[[7.999999951380232, 6.000099261882497],
 [6.000099261882497, -1.4210854715202004e-06]]
```

So everything works well, but the result is approximate, and computing the gradients of a function with regards to n variables requires calling that function n times. In deep neural nets, there are often thousands of parameters to tweak using gradient descent (which requires computing the gradients of the loss function with regards to each of these parameters), so this approach would be much too slow.

Implementing a Toy Computation Graph

Rather than this numerical approach, let's implement some symbolic autodiff techniques. For this, we will need to define classes to represent constants, variables and operations.

In [12]:

```
class Const(object):
    def __init__(self, value):
        self.value = value
    def evaluate(self):
        return self.value
    def __str__(self):
        return str(self.value)

class Var(object):
    def __init__(self, name, init_value=0):
        self.value = init_value
        self.name = name
    def evaluate(self):
        return self.value
    def __str__(self):
        return self.name

class BinaryOperator(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

class Add(BinaryOperator):
    def evaluate(self):
        return self.a.evaluate() + self.b.evaluate()
    def __str__(self):
        return "{} + {}".format(self.a, self.b)

class Mul(BinaryOperator):
    def evaluate(self):
        return self.a.evaluate() * self.b.evaluate()
    def __str__(self):
        return "({}) * ({})".format(self.a, self.b)
```

Good, now we can build a computation graph to represent the function f:

In [13]:

```
x = Var("x")
y = Var("y")
f = Add(Mul(Mul(x, x), y), Add(y, Const(2))) # f(x,y) = x²y + y + 2
```

And we can run this graph to compute f at any point, for example $f(3,4)$.

In [14]:

```
x.value = 3
y.value = 4
f.evaluate()
```

Out[14]:

```
42
```

Perfect, it found the ultimate answer.

Computing gradients

The autodiff methods we will present below are all based on the *chain rule*.

Suppose we have two functions u and v, and we apply them sequentially to some input x, and we get the result z. So we have $z = v(u(x))$, which we can rewrite as $z = v(s)$ and $s = u(x)$. Now we can apply the chain rule to get the partial derivative of the output z with regards to the input x:

$$\dfrac{\partial z}{\partial x} = \dfrac{\partial s}{\partial x} \cdot \dfrac{\partial z}{\partial s}$$

Now if z is the output of a sequence of functions which have intermediate outputs $s_1, s_2, \dots, s_n$, the chain rule still applies:

$$\frac{\partial z}{\partial x} = \frac{\partial s_1}{\partial x} \cdot \frac{\partial s_2}{\partial s_1} \cdot \frac{\partial s_3}{\partial s_2} \cdot \dots \cdot \frac{\partial s_{n-1}}{\partial s_{n-2}} \cdot \frac{\partial s_n}{\partial s_{n-1}} \cdot \frac{\partial z}{\partial s_n}$$

In forward mode autodiff, the algorithm computes these terms "forward" (i.e., in the same order as the computations required to compute the output z), that is from left to right: first $\frac{\partial s_1}{\partial x}$, then $\frac{\partial s_2}{\partial s_1}$, and so on. In reverse mode autodiff, the algorithm computes these terms "backwards", from right to left: first $\frac{\partial z}{\partial s_n}$, then $\frac{\partial s_n}{\partial s_{n-1}}$, and so on.

For example, suppose you want to compute the derivative of the function $z(x) = \sin(x^2)$ at x=3, using forward mode autodiff. The algorithm would first compute the partial derivative $\frac{\partial s_1}{\partial x} = \frac{\partial x^2}{\partial x} = 2x = 6$. Next, it would compute $\frac{\partial z}{\partial x} = \frac{\partial s_1}{\partial x} \cdot \frac{\partial z}{\partial s_1} = 6 \cdot \frac{\partial \sin(s_1)}{\partial s_1} = 6 \cdot \cos(s_1) = 6 \cdot \cos(3^2) \approx -5.46$.

Let's verify this result using the `gradients()` function defined earlier:

In [15]:

```
from math import sin

def z(x):
    return sin(x**2)

gradients(z, [3])
```

Out[15]:

```
[-5.46761419430053]
```

Look good. Now let's do the same thing using reverse mode autodiff. This time the algorithm would start from the right hand side so it would compute $\frac{}{\partial s_1} = \frac{\partial \sin(s_1)}{\partial s_1} = \cos(s_1) = \cos(3^2) \approx -0.91$. Next it would compute $\frac{}{\partial x} = \frac{\partial s_1}{\partial x} \cdot \frac{\partial z}{\partial s_1} \approx \frac{\partial s_1}{\partial x} \cdot -0.91 = \frac{\partial x^2}{\partial x} \cdot -0.91 = 2x \cdot -0.91 = 6 \cdot -0.91 = -5.46$.

Of course both approaches give the same result (except for rounding errors), and with a single input and output they involve the same number of computations. But when there are several inputs or outputs, they can have very different performance. Indeed, if there are many inputs, the right-most terms will be needed to compute the partial derivatives with regards to each input, so it is a good idea to compute these right-most terms first. That means using reverse-mode autodiff. This way, the right-most terms can be computed just once and used to compute all the partial derivatives. Conversely, if there are many outputs, forward-mode is generally preferable because the left-most terms can be computed just once to compute the partial derivatives of the different outputs. In Deep Learning, there are typically thousands of model parameters, meaning there are lots of inputs, but few outputs. In fact, there is generally just one output during training: the loss. This is why reverse mode autodiff is used in TensorFlow and all major Deep Learning libraries.

There's one additional complexity in reverse mode autodiff: the value of s_i is generally required when computing $\frac{\partial s_{i+1}}{\partial s_i}$, and computing s_i requires first computing s_{i-1}, which requires computing s_{i-2}, and so on. So basically, a first pass forward through the network is required to compute s_1, s_2, s_3, $\dots$, s_{n-1} and s_n, and then the algorithm can compute the partial derivatives from right to left. Storing all the intermediate values s_i in RAM is sometimes a problem, especially when handling images, and when using GPUs which often have limited RAM: to limit this problem, one can reduce the number of layers in the neural network, or configure TensorFlow to make it swap these values from GPU RAM to CPU RAM. Another approach is to only cache every other intermediate value, s_1, s_3, s_5, $\dots$, s_{n-4}, s_{n-2} and s_n. This means that when the algorithm computes the partial derivatives, if an intermediate value s_i is missing, it will need to recompute it based on the previous intermediate value s_{i-1}. This trades off CPU for RAM (if you are interested, check out this paper).

Forward mode autodiff

In [16]:

```
Const.gradient = lambda self, var: Const(0)
Var.gradient = lambda self, var: Const(1) if self is var else Const(0)
Add.gradient = lambda self, var: Add(self.a.gradient(var), self.b.gradient(var))
Mul.gradient = lambda self, var: Add(Mul(self.a, self.b.gradient(var)), Mul(self.a.gradie
nt(var), self.b))

x = Var(name="x", init_value=3.)
y = Var(name="y", init_value=4.)
f = Add(Mul(Mul(x, x), y), Add(y, Const(2))) # f(x,y) = x²y + y + 2

dfdx = f.gradient(x)  # 2xy
dfdy = f.gradient(y)  # x² + 1
```

In [17]:

```
dfdx.evaluate(), dfdy.evaluate()
```

Out[17]:

```
(24.0, 10.0)
```

Since the output of the `gradient()` method is fully symbolic, we are not limited to the first order derivatives, we can also compute second order derivatives, and so on:

In [18]:

```
d2fdxdx = dfdx.gradient(x) # 2y
d2fdxdy = dfdx.gradient(y) # 2x
d2fdydx = dfdy.gradient(x) # 2x
d2fdydy = dfdy.gradient(y) # 0
```

In [19]:

```
[[d2fdxdx.evaluate(), d2fdxdy.evaluate()],
 [d2fdydx.evaluate(), d2fdydy.evaluate()]]
```

Out[19]:

```
[[8.0, 6.0], [6.0, 0.0]]
```

Note that the result is now exact, not an approximation (up to the limit of the machine's float precision, of course).

Forward mode autodiff using dual numbers

A nice way to apply forward mode autodiff is to use [dual numbers](). In short, a dual number z has the form $z = a + b\epsilon$, where a and b are real numbers, and ϵ is an infinitesimal number, positive but smaller than all real numbers, and such that $\epsilon^2 = 0$. It can be shown that $f(x + \epsilon) = f(x) + \frac{\partial f}{\partial x}\epsilon$, so simply by computing $f(x + \epsilon)$ we get both the value of $f(x)$ and the partial derivative of f with regards to x.

Dual numbers have their own arithmetic rules, which are generally quite natural. For example:

Addition

$$(a_1 + b_1\epsilon) + (a_2 + b_2\epsilon) = (a_1 + a_2) + (b_1 + b_2)\epsilon$$

Subtraction

$$(a_1 + b_1\epsilon) - (a_2 + b_2\epsilon) = (a_1 - a_2) + (b_1 - b_2)\epsilon$$

Multiplication

$$(a_1 + b_1\epsilon) \times (a_2 + b_2\epsilon) = (a_1 a_2) + (a_1 b_2 + a_2 b_1)\epsilon + b_1 b_2 \epsilon^2 = (a_1 a_2) + (a_1 b_2 + a_2 b_1)\epsilon$$

Division

$$\frac{a_1 + b_1\epsilon}{a_2 + b_2\epsilon} = \frac{a_1 + b_1\epsilon}{a_2 + b_2\epsilon} \cdot \frac{a_2 - b_2\epsilon}{a_2 - b_2\epsilon} = \frac{a_1 a_2 + (b_1 a_2 - a_1 b_2)\epsilon - b_1 b_2 \epsilon^2}{{a_2}^2 + (a_2 b_2 - a_2 b_2)\epsilon - {b_2}^2\epsilon} = \frac{a_1}{a_2} + \frac{a_1 b_2 - b_1 a_2}{{a_2}^2}\epsilon$$

Power

$$(a + b\epsilon)^n = a^n + (na^{n-1}b)\epsilon$$

etc.

Let's create a class to represent dual numbers, and implement a few operations (addition and multiplication). You can try adding some more if you want.

In [20]:

```
class DualNumber(object):
    def __init__(self, value=0.0, eps=0.0):
        self.value = value
        self.eps = eps
    def __add__(self, b):
        return DualNumber(self.value + self.to_dual(b).value,
                          self.eps + self.to_dual(b).eps)
    def __radd__(self, a):
        return self.to_dual(a).__add__(self)
    def __mul__(self, b):
        return DualNumber(self.value * self.to_dual(b).value,
                          self.eps * self.to_dual(b).value + self.value * self.to_dual(b
).eps)
    def __rmul__(self, a):
        return self.to_dual(a).__mul__(self)
    def __str__(self):
        if self.eps:
            return "{:.1f} + {:.1f}ε".format(self.value, self.eps)
        else:
            return "{:.1f}".format(self.value)
    def __repr__(self):
        return str(self)
    @classmethod
    def to_dual(cls, n):
        if hasattr(n, "value"):
            return n
        else:
            return cls(n)
```

$3 + (3 + 4\epsilon)$
$= 6 + 4\epsilon$

In [21]:

```
3 + DualNumber(3, 4)
```

Out[21]:

```
6.0 + 4.0ε
```

$(3 + 4\varepsilon) \times (5 + 7\varepsilon)$ = $3 \times 5 + 3 \times 7\varepsilon + 4\varepsilon \times 5 + 4\varepsilon \times 7\varepsilon$ = $15 + 21\varepsilon + 20\varepsilon + 28\varepsilon^2$ = $15 + 41\varepsilon + 28 \times 0$ = $15 + 41\varepsilon$

In [22]:

```
DualNumber(3, 4) * DualNumber(5, 7)
```

Out[22]:

```
15.0 + 41.0ε
```

Now let's see if the dual numbers work with our toy computation framework:

In [23]:

```
x.value = DualNumber(3.0)
y.value = DualNumber(4.0)

f.evaluate()
```

Out[23]:

```
42.0
```

Yep, sure works. Now let's use this to compute the partial derivatives of f with regards to x and y at x=3 and y=4:

In [24]:

```
x.value = DualNumber(3.0, 1.0)   # 3 + ε
y.value = DualNumber(4.0)        # 4

dfdx = f.evaluate().eps

x.value = DualNumber(3.0)        # 3
y.value = DualNumber(4.0, 1.0)   # 4 + ε

dfdy = f.evaluate().eps
```

In [25]:

```
dfdx
```

Out[25]:

```
24.0
```

In [26]:

```
dfdy
```

Out[26]:

```
10.0
```

Great! However, in this implementation we are limited to first order derivatives. Now let's look at reverse mode.

Reverse mode autodiff

Let's rewrite our toy framework to add reverse mode autodiff:

In [27]:

```
class Const(object):
    def __init__(self, value):
        self.value = value
    def evaluate(self):
        return self.value
    def backpropagate(self, gradient):
        pass
    def __str__(self):
        return str(self.value)

class Var(object):
    def __init__(self, name, init_value=0):
        self.value = init_value
        self.name = name
        self.gradient = 0
    def evaluate(self):
        return self.value
    def backpropagate(self, gradient):
        self.gradient += gradient
    def __str__(self):
        return self.name

class BinaryOperator(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

class Add(BinaryOperator):
    def evaluate(self):
        self.value = self.a.evaluate() + self.b.evaluate()
        return self.value
    def backpropagate(self, gradient):
        self.a.backpropagate(gradient)
        self.b.backpropagate(gradient)
    def __str__(self):
        return "{} + {}".format(self.a, self.b)

class Mul(BinaryOperator):
    def evaluate(self):
        self.value = self.a.evaluate() * self.b.evaluate()
        return self.value
    def backpropagate(self, gradient):
        self.a.backpropagate(gradient * self.b.value)
        self.b.backpropagate(gradient * self.a.value)
    def __str__(self):
        return "({}) * ({})".format(self.a, self.b)
```

In [28]:

```
x = Var("x", init_value=3)
y = Var("y", init_value=4)
f = Add(Mul(Mul(x, x), y), Add(y, Const(2))) # f(x,y) = x²y + y + 2

result = f.evaluate()
f.backpropagate(1.0)
```

In [29]:

```
print(f)
```

```
((x) * (x)) * (y) + y + 2
```

In [30]:

```
result
```

Out[30]:

```
42
```

In [31]:

```
x.gradient
```

Out[31]:

```
24.0
```

In [32]:

```
y.gradient
```

Out[32]:

```
10.0
```

Again, in this implementation the outputs are just numbers, not symbolic expressions, so we are limited to first order derivatives. However, we could have made the `backpropagate()` methods return symbolic expressions rather than values (e.g., return `Add(2,3)` rather than 5). This would make it possible to compute second order gradients (and beyond). This is what TensorFlow does, as do all the major libraries that implement autodiff.

Reverse mode autodiff using TensorFlow

In [33]:

```
import tensorflow as tf
```

In [34]:

```
x = tf.Variable(3.)
y = tf.Variable(4.)

with tf.GradientTape() as tape:
    f = x*x*y + y + 2

jacobians = tape.gradient(f, [x, y])
jacobians
```

Out[34]:

```
[<tf.Tensor: shape=(), dtype=float32, numpy=24.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=10.0>]
```

Since everything is symbolic, we can compute second order derivatives, and beyond:

In [35]:

```
x = tf.Variable(3.)
y = tf.Variable(4.)

with tf.GradientTape(persistent=True) as tape:
    f = x*x*y + y + 2
    df_dx, df_dy = tape.gradient(f, [x, y])

d2f_d2x, d2f_dydx = tape.gradient(df_dx, [x, y])
d2f_dxdy, d2f_d2y = tape.gradient(df_dy, [x, y])
del tape

hessians = [[d2f_d2x, d2f_dydx], [d2f_dxdy, d2f_d2y]]
hessians
```

```
WARNING:tensorflow:Calling GradientTape.gradient on a persistent tape inside its context
is significantly less efficient than calling it outside the context (it causes the gradie
nt ops to be recorded on the tape, leading to increased CPU and memory usage). Only call
GradientTape.gradient inside the context if you actually want to trace the gradient in or
der to compute higher order derivatives.
```

Out[35]:

```
[[<tf.Tensor: shape=(), dtype=float32, numpy=8.0>,
  <tf.Tensor: shape=(), dtype=float32, numpy=6.0>],
 [<tf.Tensor: shape=(), dtype=float32, numpy=6.0>, None]]
```

Note that when we compute the derivative of a tensor with regards to a variable that it does not depend on, instead of returning 0.0, the `gradient()` function returns `None`.

And that's all folks! Hope you enjoyed this notebook.