Understanding Autodiff with JAX

17 November 2020

Automatic differentiation, or autodiff is a set of techniques for computing the gradient of a function using the chain rule of differentiation and is at the core of various deep learning frameworks like PyTorch, TensorFlow and Theano. JAX is one such framework that can perform autodiff on functions defined in native Python or NumPy code and provides other transformation that make gradient-based optimizations easy and intuitive. This post attempts to understand the mechanism of autodiff while working with JAX.

1. Gradient of vector-valued function

Partial derivative of a function that depends on multiple variables is the derivative with respect to one of those variables while keeping others constant. Partial derivatives are used to define the gradient for vector-valued functions.

Consider function $F : \mathbb{R}^n \to \mathbb{R}^m$ and vector $X = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^T \in \mathbb{R}^n$.

$$F(X) = \begin{bmatrix} f_1(X) & \cdots & f_m(X) \end{bmatrix}^T \in \mathbb{R}^m$$

The first-order gradient, called the Jacobian, is a matrix of first-order partial derivatives.

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

For example, consider the following function.

$$F(X) = \begin{bmatrix} f_1(X) \\ f_2(X) \end{bmatrix} = \begin{bmatrix} x_1 x_2^2 \\ x_1 x_2 \end{bmatrix}$$

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2^2 & 2x_1x_2 \\ x_2 & x_1 \end{bmatrix}$$

Let’s define the same function in Python using NumPy and use JAX to compute Jacobian at $x = \begin{bmatrix} 3 & 4 \end{bmatrix}^T$

```python
import jax.numpy as jnp
from jax import jacrev


def f(x):
    x1, x2 = x[0], x[1]
    return jnp.array([x1*x2*x2, x1*x2])


J = jacrev(f)

x = jnp.array([3.0, 4.0])
print(J(x))

'''
Output:

[[16. 24.]
 [ 4.  3.]]
 '''
```

jacobian_example.py hosted with ❤ by GitHub view raw

Notice that the output matches the analytical result we derived above. We use the JAX NumPy APIs which are extended to support autodiff. To compute the Jacobian, we use the function called `jacrev` which is short for `Jacobian Reverse`. We will get to the significance of the term “reverse” later.

2. Autodiff

Autodiff computes the gradient of a function by breaking it down into primitive functions, computing their gradients and applying chain rule. It is different from other methods of computing gradients like numerical approximation (tending the difference to a small finite value) or symbolic differentiation (deriving formula for gradient using a symbolic library).

Consider a function,

$$y = F(x) = C(B(A(x)))$$

Let the intermediate values be, $a = A(x), b = B(a)$ and $y = C(b)$.

Using chain rule,

$$\frac{dy}{dx} = \frac{dy}{db}\frac{db}{da}\frac{da}{dx}$$

We are able to compute the gradient of F by computing the gradient values of the primitive functions A, B, C. But how do we compute the gradients for the primitive functions? We break down F till the primitive functions A, B, C are known elementary functions (like addition, multiplication, `sin`, `cos`, `tanh`) for which we compute gradient using efficient routines provided by our autodiff library. By composing the Jacobian matrices of the primitive functions, using chain rule, we compute the gradient of F.

Please note that even though in this case the chain rule results in a literal chain where we only have to multiply the individual gradient matrices, it need not be the case always (it is a DAG generally). For example, consider $F(x) = x^2 sin(x)$.

Forward Mode and Reverse Mode

But for now, lets come back to $\frac{dy}{dx} = \frac{dy}{db}\frac{db}{da}\frac{da}{dx}$. Due to associativity of matrix multiplication, the order in which we multiply the gradients does not matter for the final result. But the order becomes important when we need to do it efficiently. The problem of finding the most efficient way to compute the product of a chain of matrices has a solution using dynamic programming. But lets only consider the two extreme cases.

This is called the forward mode (move from input side to output side),

$$\frac{dy}{dx} = \frac{dy}{db}\left(\frac{db}{da}\frac{da}{dx}\right)$$

And this is the reverse mode (move from output side to input side),

$$\frac{dy}{dx} = \left(\frac{dy}{db}\frac{db}{da}\right)\frac{da}{dx}$$

In practice, the forward and backward modes for computing Jacobian are implemented using Jacobian-Vector Product and Vector-Jacobian Product respectively.

For function $y = F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, a Jacobian-Vector Product is,

$$\frac{dy}{dx}v = \frac{dy}{db}\left(\frac{db}{da}\left(\frac{da}{dx}v\right)\right), v \in \mathbb{R}^n$$

and a Vector-Jacobian Product is computed as,

$$v^T\frac{dy}{dx} = \left(\left(v^T\frac{dy}{db}\right)\frac{db}{da}\right)\frac{da}{dx}, v \in \mathbb{R}^m$$

The use of JVP (in case of forward mode) and VJP (in case of backward mode) is more memory efficient than computing the Jacobian directly. Multiplication by a vector ensures that the intermediate results of the chain rule are all 1D vectors that are easy on machine memory. In case of JVP, in forward mode, by setting v as one-hot basis vector ($[0 \quad \cdots \quad 1 \quad \cdots \quad 0]^T$), we get a single COLUMN of the Jacobian. Similarly, in case of VJP in reverse mode, a one-hot basis vector produces a single ROW of the Jacobian. Through multiple passes of the JVP or VJP, we can form the entire Jacobian.

Here’s a piece of code that shows the computation of Jacobian by stacking VJPs of basis vectors.

```python
import jax.numpy as jnp
from jax import random, jacrev, vjp

key = random.PRNGKey(0)


def sigmoid(x):
    return 0.5 * (jnp.tanh(x / 2) + 1)


def predict(W, b, inputs):
    return sigmoid(jnp.dot(inputs, W) + b)


key, W_key, b_key = random.split(key, 3)
W = random.normal(W_key, (3,))
```

```python
b = random.normal(b_key, ())

inputs = jnp.array([[0.52, 1.12,  0.77],
                    [0.88, -1.08, 0.15],
                    [0.52, 0.06, -1.30],
                    [0.74, -2.49, 1.39]])


def f(W):
    return predict(W, b, inputs)


def basis(size, index):
    a = [0.0] * size
    a[index] = 1.0
    return jnp.array(a)


M = [basis(4, i) for i in range(0, 4)]

# computing by stacking VJPs of basis vectors
y, vjp_fun = vjp(f, W)

print('Jacobian using vjp and stacking:')
print(jnp.vstack([vjp_fun(mi) for mi in M]))

# computing directly using jacrev function
print('Jacobian using jacrev directly:')
print(jacrev(f)(W))


'''
Output:

Jacobian using vjp and stacking:
[[ 0.05981752  0.12883773  0.08857594]
 [ 0.04015911 -0.04928619  0.0068453 ]
 [ 0.12188289  0.01406341 -0.3047072 ]
 [ 0.00140426 -0.00472514  0.00263773]]
Jacobian using jacrev directly:
[[ 0.05981752  0.12883773  0.08857594]
 [ 0.04015911 -0.04928619  0.0068453 ]
 [ 0.12188289  0.01406341 -0.3047072 ]
 [ 0.00140426 -0.00472514  0.00263773]]
 '''
```

jacrev.py hosted with ❤ by GitHub view raw `jacrev` (reverse mode Jacobian) and `jacfwd` (forward mode Jacobian) are implemented using `vjp` and `jvp` respectively. In the actual implementation in JAX, they do not loop over the basis vectors and stack the Jacobian. Instead, `vmap` is used to vectorize the whole operation and is much faster.

JVP versus VJP

Depending on the shape of x and y, either of them could be more efficient than the other. For $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, if $m >> n$, the Jacobian has more rows than columns and hence assembling Jacobian column-by-column using JVP is more efficient. If $n >> m$, there are more columns than rows, and reverse mode using VJP is preferred.

Often in optimization problems, the function tends to map from high-dimensional input to a scalar output (like a convolutional neural network mapping an image to a loss value). Since $n >> m$ in those cases, it is considerably more efficient to use reverse mode autodiff, also called backpropogation.

3. JAX

JAX has turned out to be a formidable framework due to its ability to differentiate functions written in native Python and NumPy code including loops, ifs, recursions and exceptions. JAX uses XLA to compile and run the program on hardware accelerators like GPUs and TPUs.

```python
from jax import grad
import jax.numpy as jnp


def f(x):
    if x < 5:
        y = 0
        for i in range(0, 3):
            y += x**i
        return y
    else:
        return 2*x


grad_func = grad(f)  # grad returns gradient fn for fn with scalar output

print('f(3) = {}, f\'(3) = {}'.format(f(3), grad_func(3.)))
print('f(8) = {}, f\'(8) = {}'.format(f(8), grad_func(8.)))

'''
Output:
f(3) = 13, f'(3) = 7.0
f(8) = 16, f'(8) = 2.0
'''
```

autodiff_example.py hosted with ❤ by GitHub view raw

JAX provides APIs for computing JVP (`jvp`) and VJP (`vjp`) and they are respectively used in the implementation of `jacfwd` and `jacrev`. Other JAX APIs like `jit` and `vmap` are incredibly powerful but we will not deal with in this tutorial.

Autograd

JAX implements autodiff using a library called Autograd. During function call, Autograd records all the primitive functions applied on the input to create a computational graph. After the function is evaluated, the computational graph is parsed and VJP/JVP functions are generated for each primitive function using a mapping from primitives to their corresponding VJP/JVP functions. The VJPs/JVPs are composed using chain rule to compute the gradient as discussed above.

Since Autograd records the operations on every input separately, it can easily support any sort of control flow in the Python code. This simplifies the implementation of autodiff compared to other frameworks like TensorFlow Graphs where we have to define the computational graph using a limited set of nodes provided by the framework.

Let the element-wise function $b = \tanh(a)$ be one of the primitive functions in a backpropogation chain.

$$v^T \frac{dy}{dx} = \left(\underbrace{\left(v^T \frac{dy}{db} \right)}_{1 \times k} \underbrace{\frac{db}{da}}_{k \times k} \right) \frac{da}{dx}$$

Since it is an element-wise operation, the Jacobian will be a square matrix of size $k \times k$ where $a \in \mathbb{R}^k$. Further, the Jacobian of an element-wise operation would be a diagonal matrix since $\frac{\partial b_i}{\partial a_j} = 0, i \neq j$. We can use this property to compute VJP without actually forming the $k \times k$ Jacobian.

```python
import numpy as np


def make_vjp_tanh(input, output):
    return lambda g: g/np.cosh(input)**2
```

vjp_tanh.py hosted with ❤ by GitHub view raw

In the code, `input` and `output` are respectively the input vector (a) and output vector (b) to the $\tanh$ function. g is the $1 \times k$ VJP that has accumulated gradients from left up until the current function. Instead of matrix multiplying g by the entire diagonal Jacobian matrix, we multiply elements of g with corresponding diagonal elements of the Jacobian computed using the differentiation rule.

$$(\tanh(x))' = \frac{1}{\cosh^2(x)}$$

4. Example

Combining the ideas explored so far, we can write our own basic autodiff using only NumPy functions. Here's a small code snippet that computes the gradient for the element-wise function $f(x) = e^{\tanh(x)}$. To verify the correctness, we also compute the Jacobian using JAX.

```python
import jax.numpy as jnp
from jax import random, jacrev

key = random.PRNGKey(0)


def make_vjp_tanh(input, output):
    return lambda g: g/jnp.cosh(input)**2


def make_vjp_exp(input, output):
    return lambda g: g * output


def f1(x):
    return jnp.tanh(x)


def f2(x):
    return jnp.exp(x)

```

```python

x = random.normal(key, (3,))

x1 = f1(x)
vjp_f1 = make_vjp_tanh(x, x1)

y = f2(x1)
vjp_f2 = make_vjp_exp(x1, y)


def vjp(v): return vjp_f1(vjp_f2(v))  # chain rule


# compute each row of Jacobian using vjp and combine
v1 = jnp.array([1, 0, 0])
v2 = jnp.array([0, 1, 0])
v3 = jnp.array([0, 0, 1])

print('Jacobian: ')
print(jnp.stack([vjp(v1), vjp(v2), vjp(v3)], axis=0))

# now we do the same with JAX


def f(x): return f2(f1(x))


print('Jacobian using JAX: ')
print(jacrev(f)(x))

'''
Output:

Jacobian:
[[0.25932845 0.         0.        ]
 [0.         0.51030356 0.        ]
 [0.         0.         1.2386373 ]]
Jacobian using JAX:
[[0.25932842 0.         0.        ]
 [0.         0.5103035  0.        ]
 [0.         0.         1.2386374 ]]
'''
```

vjp.py hosted with ❤ by GitHub view raw

5. More

1. Matthew Johnson’s tutorial on autodiff

2. JAX autodiff cookbook

3. Autodiff notes

4. Autodidact: a pedagogical implementation of Autograd

← Back